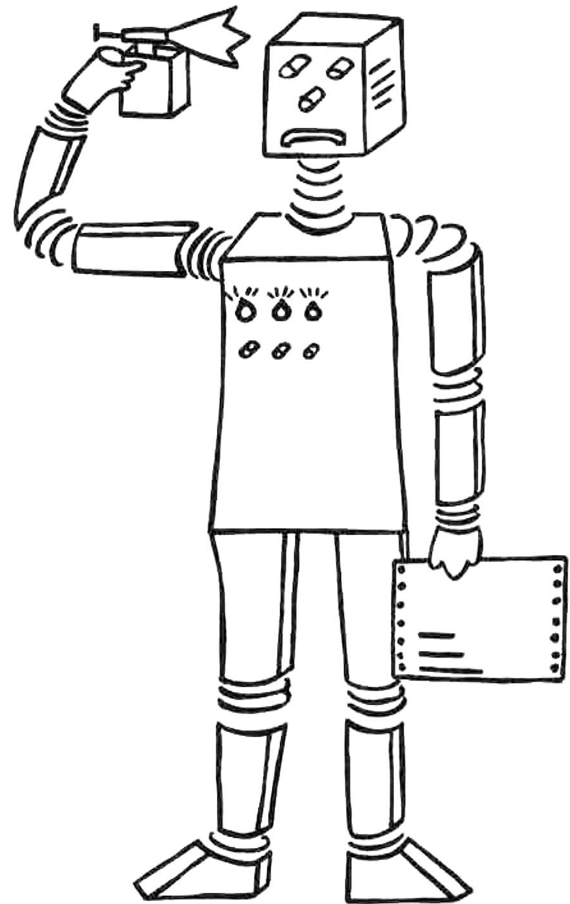
Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern 1, W. Germany

Artificial
Intelligence
Laboratories

SEKI-REPORT



On implementations of loose abstract data type specifications and their vertical composition

Christoph Beierle, Angelika Voß

August 1986 SEKI-REPORT SR-86-14

On implementations of loose abstract data type specifications and their vertical composition

Christoph Beierle, Angelika Voß
Fachbereich Informatik, Universität Kaiserslautern
Postfach 3049, 6750 Kaiserslautern, West Germany
UUCP: ..!mcvax!unido!uklirb!beierle

Abstract

In an approach for the implementation of loose abstract data type specifications that completely distinguishes between the syntactical level of specifications and the semantical level of models, vertical implementation composition is defined compatibly on both levels. Implementations have signatures, models, and sentences where the latter also include hidden components, which allows for useful normal form results. We illustrate the stepwise development of implementations as well as their composition by some examples and describe the incorporation of the concept into an integrated software development and verification system.

Contents

1. Introduction

2. Implementation specifications: Basic idea and requirements for their composition

3. Implementation specifications without hidden components

4. Implementation specifications with hidden components

5. Normal forms

6. Composition

7. Implementation specifications in the ISDV system

8. Conclusions

1. Introduction

In the early days of abstract data types merely fixed ADT specifications with only isomorphic models were studied. Later on, so-called loose approaches were suggested where one considers not only the initial or terminal model of a specification but all models. As one of its main advantages a loose approach is better suited to capture the process of software development: One can start with a small and still vague specification with many different models, and then refine such a specification gradually by adding new axioms, sorts, and operations, thereby restricting the class of admissible models. During this process, lower level constructive definition techniques may be used to refine the higher level axiomatic definitions so that one finally arrives at a concrete problem solution, which could be a program or a functional prototype.

An implementation relation between loose specifications should reflect this refinement scenario: among the many different models of the source and target specification one should be able to choose those of interest by gradually refining the implementation so that the set of models is restricted accordingly. Our implementation concept introduced in [BV 85a] generalizes the concept for implementations of loose specifications proposed by Sannella and Wirsing in [SW 82], which in turn generalizes the fixed case (e.g. [GTW 78], [Ehc 82], [EKP 78], [EKMP 82], [Ga 83]). By using the notion of institution ([GB 83]) our approach abstracts from the types of sentences used in the underlying ADT specification method.

One of the central problems when dealing with implementations is their composability. In this paper we show how our concept of implementation specifications allows for a two level approach: The composition of implementations is defined both on the syntactical level of specifications and on the semantical level of models. Both levels are closed under their composition operations which are associative. In particular, by using a strong normal result we show that syntactical and semantical compositions are compatible with each other.

In Section 2 we summarize the basic idea of our implementation concept as given in [BV 85a], elaborate the requirements a composition operation should fulfill, and briefly state the assumptions about the underlying loose ADT specifications. In Section 3 we introduce the institution of implementation specifications without hidden components, and in Section 4 we extend this institution by introducing hidden parts. Section 5 contains our normal form theorem, and in Section 6 we develop syntactical and semantical composition operations and show their compatibility. Section 7 describes the incorporation of our concept into an integrated software development and verification system, and Sect. 8 contains a summary and a comparison.

2. Implementation specifications: Basic idea and requirements for their composition

As compared to fixed specifications, in the loose case we still have specifications, signatures, signature morphisms, etc, the essential difference lying in the number of models being considered. Therefore, an implementation for loose specifications should at least consist of an abstract specification, a concrete specification, and a signature morphism translating the abstract signature to the (possibly extended) concrete signature. Since a concrete specification can always be extended before giving the implementation, we will choose the technically simpler approach and omit any extension of the concrete specification as part of the implementation.
In [SW 82] Sannella and Wirsing require that for every concrete model there should be some abstract model and an abstraction function connecting them. If such a complete set of triples exists, the concrete specification is said to implement the abstract one, otherwise it does not. This is an implicit, non-constructive approach which gives no room for a notion of refinement between implementations since there is no way to characterize and restrict the set of triples - e.g. by constraints on the concrete or abstract models - any further.
As already pointed out above, since the idea of loose specifications is to consider at first

an arbitrarily large set of models and to restrict this set stepwise by refining the specification, we think the adequate idea of implementations between loose specifications is to accept all meaningful combinations of an abstract model, a concrete model, and an abstraction function and to restrict them stepwise by refining the implementation.

To realize these ideas we introduce the notion of implementation models:

A simple implementation <SPa,σ,SPc> consisting of an abstract specification SPa, a concrete one SPc, and a signature translation σ between them denotes the set of all triples consisting of an abstract model Aa, a concrete one Ac, and an abstraction function α from the concrete to the abstract model. Such a tripel <Ac,α,Aa> is called an implementation model. As in the fixed case, the abstraction function may be partially defined and it must be surjective and homomorphic.

(Note that in both cases the first component contains the source and the third component the target of the function in the middle component.)

Now we extend these simple implementations to a concept incorporating a notion of refinement between implementations. Such a refinement should restrict the set of implementation models which can be done componentwise by

- restricting the set of abstract models,
- restricting the set of concrete models,
- restricting the set of abstraction functions.

In the framework of loose specifications the set of models - like the abstract and the concrete ones - is restricted by adding sentences to the respective specification.

Since the abstraction functions operate on both concrete and abstract carriers we propose to view them as algebra operations from concrete to abstract sorts. These operations can be restricted as usually by adding sentences over both the concrete and the abstract signatures extended by the abstraction operation names. Thus in a first approach we admit arbitrary sentences over the abstract and the concrete signatures extended by the abstraction operation names, and later on we will extend this vocabulary by arbitrary hidden sorts and operations. These sentences will be called implementation sentences.

Summarizing we propose an implementation specification ISP = <IΣ, IE> to be

- a simple implementation IΣ = <SPa,σ,SPc>
- together with a set of implementation sentences IE and
- denoting all implementation models of the simple implementation which satisfy the implementation sentences.

Analogously to specifications which consist of a signature in the simplest case, a simple implementation like IΣ will also be called an implementation signature.

We already claimed that an implementation should be refinable by adding more implementation sentences to it and thus reducing the class of implementation models. This idea is extended analogously to loose ADT specifications by admitting a change of signature: There, a specification morphism is a signature morphism such that the translated sentences of the refined specification hold in the refining specification. Thus an implementation morphism or a refinement between two implementations is an implementation signature morphism such that the translated sentences of the refined implementation hold in the refining one.

Since an implementation signature $I\Sigma_j$ = <SPaj,σj,SPcj> contains two specifications an

implementation-morphism is a pair

$$\tau = \langle \rho a, \rho c \rangle : I\Sigma_1 \rightarrow I\Sigma_2$$

consisting of an abstract specification morphism $\rho a: SPa_1 \rightarrow SPa_2$ and a concrete specification morphism $\rho c: SPc_1 \rightarrow SPc_2$.

However, another requirement should also be satisfied: Assume we have an implementation from sets over arbitrary elements to extended lists over arbitrary elements, and another i-signature from sets over natural numbers to extended lists over natural numbers. Then it should not matter whether we first represent sets over arbitrary elements by lists over arbitrary elements and then refine to lists over natural numbers, or if we first refine the sets over arbitrary elements to sets over natural numbers and then represent them as lists over natural numbers. In general this means that the diagram

$$\begin{array}{ccc} SPa_1 & \xrightarrow{\sigma_1} & SPc_1 \\ \rho a \downarrow & & \downarrow \rho c \\ SPa_2 & \xrightarrow{\sigma_2} & SPc_2 \end{array}$$

should commute viewing ρa and ρc as signature morphisms.

When implementing an abstract specification by a more concrete one which in turn is implemented by a third specification it is desirable to get automatically an implementation of the first by the third specification by <u>composing</u> the two individual implementations. Moreover, the sequence of compositions should be irrelevant, i.e. one would like to have an associative implementation composition operator.

In the most elaborated implementation concept for the fixed case given in [EKMP 82], proof theoretical and semantical conditions are given that guarantee composability. In the loose approach of [SW 82], full composability is given by the very definition of implementation: every concrete algebra must be associated to an abstract algebra. Further approaches studying such compositions of implementations are the approaches of [Hup 81], [GM 82], [Ga 83], [Li 83], and [SW 83].

In our concept of implementation specifications as outlined above the question of composability arises on different levels. Simple implementations, implementation specifications, and implementation models should all be composable and closed under composition.

When composing simple implementations it is natural to require that the concrete specification of the first implementation is identical to the abstract one of the second implementation. In this case the composed implementation is obtained by taking the abstract specification of the first implementation and the concrete specification of the second one together with the composition of the two signature translations.

A similar argument holds for the composition of implementation models. In particular one must ensure that the abstraction functions are composable and closed under composition.

Having defined the composition of simple implementations and implementation models, the composition of implementation specifications containing implementation sentences can be constrained by the following <u>compatibility condition</u> which arises naturally:

- The set of models of a composed implementation specification should be identical to the set of models obtained by composing the sets of models of the

individual implementation specifications.

If this condition is fulfilled we say that the composition of implementation specifications is compatible with the composition of their implementation models.

Summarizing we require our implementation concept to offer the following features w.r.t. composition:

- definition of composition for
  - simple implementations (i.e. implementation signatures),
  - implementation specifications, and
  - implementation models

such that all of them are

- closed under composition,

and such that composition is

- associative, and
- compatible w.r.t. implementation specifications and implementation models.

W.r.t. the underlying institution of loose specifications we only assume that the loose specifications have equational signatures with error constants, denote strict algebras, and are formally defined as the theories of an institution ([GB 83]). In particular, we do not make any assumptions about the types of sentences:

Assumption: SPEC-institution := $\langle$SIG, EAlg, ESen, $\models^{e}$ $\rangle$

is an institution where

- SIG is a category of equational signatures with an error constant error-s for each sort s.
- EAlg is a coproduct preserving model functor mapping a signature Σ to all strict Σ-algebras, which have flat cpos as carriers, strict operations, and the error constants denoting the bottom element.
- ESen is a sentence functor mapping a signature Σ to a set of Σ-sentences.
- $\models^{e}$ is the strict satisfaction relation.

SPEC denotes the category of theories in the SPEC-institution which will be called (loose) specifications, and Sig: SPEC $\to$ SIG is the functor forgetting specifications to their signatures.

3. Implementation specifications without hidden components

3.1 Implementation signatures

The notions of implementation signatures and morphisms as sketched in Section 2 constitute a category. In fact, it is the comma category induced by the functor Sig forgetting specifications to their signatures.

Definition 3.1 [ISIG, i-signature]

Given the forgetful functor Sig: SPEC $\to$ Sig, the comma category

ISIG = (Sig$\downarrow$Sig)

is the category of implementation signatures (i-signatures).

Since the category SIG is cocomplete and the functor Sig preserves all colimits, ISIG

is cocomplete, too, by a general property of comma categories.

Fact 3.2 ⌊colimits⌋ ISIG is cocomplete.

3.2 Implementation models

According to Section 2 we want to introduce abstraction operations as ordinary operations which are interpreted by abstraction functions and which can be restricted by ordinary sentences. Since in the framework of the SPEC-institution the algebra operations must be totally defined, we will also require that the abstraction operations are totally defined. This is no limitation because the algebras are cpos and there is an error constant for each sort denoting the minimum element. Thus $\alpha(x)$ is mapped to error whenever $\alpha(x)$ is meant to be undefined. Doing so we must only suitably restrict the homomorphism requirement

$$\alpha(\sigma(op)(x)) = op(\alpha(x))$$

which under these circumstances needs to hold only if $\alpha(x)$ is non-error.

Definition 3.3 ⌊Σ-p-homomorphism⌋

Let A, B ε EAlg(Σ) with Σ = <S,Op> ε SIG. An S-sorted family of functions

$$h = \{h_s\colon A_s \to B_s \mid s \;\varepsilon\; S\}$$

is a partially-homomorphic Σ-homomorphism (or just Σ-p-homomorphism) iff

$$\forall\, op\colon s_1 \ldots s_n \to s \;\varepsilon\; \Sigma \;.$$
$$\forall\, x_1 \;\varepsilon\; A_{s1} \;. \;\ldots\; \forall\, x_n \;\varepsilon\; A_{sn} \;.$$
$$h_{s1}(x_1) \neq \text{error-}s_{1B} \;\&\; \ldots \;\&\; h_{sn}(x_n) \neq \text{error-}s_{nB}$$
$$\Rightarrow h_s(op_A(x_1,\ldots,x_n)) = op_B(h_{s1}(x_1),\ldots,h_{sn}(x_n))$$

Fact 3.4 ⌊p-homomorphisms are closed under composition⌋

Let Σ = <S,Op> ε SIG and f: A → B, g: B → C be Σ-p-homomorphisms. Then their composition

$$g \circ f := \{g_s \circ f_s \mid s \;\varepsilon\; S\}\colon A \to C$$

is a Σ-p-homomorphism.

Definition 3.5 ⌊PEAlg⌋

The functor

$$\text{PEAlg: SIG} \to \text{CAT}^{op}$$

maps a signature Σ to the category of strict Σ-algebras with Σ-p-homomorphisms, and it maps a signature morphism σ to the forgetful functor PEAlg(σ) which is defined analogously to EAlg(σ).

Fact 3.6 ⌊Partial⌋

The family of inclusion functors

$$\text{Partial}_\Sigma\colon \text{EAlg}(\Sigma) \to \text{PEAlg}(\Sigma)$$

with Σ ε SIG defines a natural transformation

$$\text{Partial: EAlg} \Longrightarrow \text{PEAlg}.$$

With PEAlg formalizing the property "partially homomorphic" we are now ready to define a preliminary model functor mapping an i-signature IΣ to the category of all tripels TA = <Ac,α,Aa> where α is p-homomorphic but not necessarily surjective. Analogously to i-signature morphisms the morphisms in this category are pairs of homomorphisms

$$\langle hc,ha\rangle : \langle Ac,\alpha,Aa\rangle \to \langle Bc,\beta,Ba\rangle$$

that are compatible with the abstraction functions, i.e. it does not matter whether we first abstract Ac-elements with α to Aa-elements and then map them with hc to Bc-elements, or whether we first map the Ac-elements with hc to Bc-elements and then abstract them with β to Ba. As in the fixed case, the forgetful functor EAlg(σ) is applied to the source of the abstraction functions so that the compatability condition for the model morphisms is the commutativity of the diagram

$$\begin{array}{ccc} \mathrm{EAlg}(\sigma)(\mathrm{Ac}) & \xrightarrow{\ \alpha\ } & \mathrm{Aa} \\ {\scriptstyle \mathrm{EAlg}(\sigma)(\mathrm{hc})}\downarrow & & \downarrow{\scriptstyle \mathrm{ha}} \\ \mathrm{EAlg}(\sigma)(\mathrm{Bc}) & \xrightarrow{\ \beta\ } & \mathrm{Ba} \end{array}$$

in PEAlg(Σa).

Similar to i-signatures, this situation can be expressed neatly as a comma category.

Definition 3.7 [Tripel(IΣ)]

Let IΣ = <SPa,σ,SPc> be an i-signature with Sig(SPa) = Σa and Sig(SPc) = Σc. The comma category

$\mathrm{Tripel}(I\Sigma) := (\mathrm{Partial}_{\Sigma a} \circ \mathrm{EAlg}(\sigma)|_{\mathrm{EAlg(SPc)}} \downarrow \mathrm{Partial}_{\Sigma a}|_{\mathrm{EAlg(SPa)}})$

is called the category of IΣ-tripels.

Similar to ordinary signatures, every i-signature morphism induces a forgetful functor between the respective model categories in the reverse direction. It is defined componentwise.

Fact 3.8 [Tripel(τ)]

Let $\tau = \langle \rho a, \rho c\rangle : I\Sigma_1 \to I\Sigma_2 \in \mathrm{ISIG}$.

$\mathrm{Tripel}(\tau): \mathrm{Tripel}(I\Sigma_2) \to \mathrm{Tripel}(I\Sigma_1)$

defined on objects by

Tripel(τ)(<Ac,α,Aa>) := <EAlg(ρc)(Ac),PEAlg(ρa)(α),EAlg(ρa)(Aa)>

and on morphisms by

Tripel(τ)(<hc,ha>) := <EAlg(ρc)(hc),EAlg(ρa)(ha)>

is a functor.

The observations above yield a prelimininary model functor Tripel: ISIG → CAT^{op}. We still have to restrict this functor to consider only tripels with surjective abstraction functions.

Definition 3.9 [IMod(IΣ)]

For every IΣ ε ISIG the category of IΣ-implementation models (or just IΣ-i-models)

IMod(IΣ)

is the full subcategory of Tripel(IΣ) generated by all tripels with surjective abstraction function.

Fact 3.10 [IMod(τ), IMod]

For every $\tau: I\Sigma_1 \to I\Sigma_2$ the restriction and corestriction of Tripel(τ) to $\mathrm{IMod}(I\Sigma_2)$ and $\mathrm{IMod}(I\Sigma_1)$ exists. It is denoted by

$\mathrm{IMod}(\tau): \mathrm{IMod}(I\Sigma_2) \to \mathrm{IMod}(I\Sigma_1)$

and

IMod: ISIG → CAT^{op}

is called the modelling functor for implementation signatures.

3.3 Relating implementation signatures to specifications

According to Section 2, implementation sentences over an i-signature $I\Sigma$ shall be expressed over the abstract signature Σa, the concrete signature Σc, and so-called abstraction operations to be interpreted as abstraction functions. In a first approach, implementation sentences will be all ordinary sentences over this vocabulary. For reasons of convenience we will use standard names for the abstraction operations:

Definition 3.11 [abs-operations]

For $I\Sigma = \langle\langle\Sigma a, Ea\rangle, \sigma, \langle\Sigma c, Ec\rangle\rangle \in ISIG$ and $\tau = \langle\rho a, \rho c\rangle: I\Sigma \to I\Sigma' \in /ISIG/$ we define:

$\text{abs-operations}(I\Sigma) := \{\text{abs-}s_{I\Sigma}: \sigma(s) \to s \mid s \in \Sigma a\}$

$\text{abs-operations}(\tau) := \{(\text{abs-}s_{I\Sigma}, \text{abs-}\rho a(s)_{I\Sigma'}) \mid s \in \Sigma a\}$.

Fact 3.12 [ψ]

$\psi: ISIG \to SIG$

defined on objects by

$\psi(I\Sigma) := \Sigma a \uplus \Sigma c \uplus \text{abs-operations}(I\Sigma)$

and on morphisms by

$\psi(\tau) := \rho a \uplus \rho c \uplus \text{abs-operations}(\tau)$

is a colimit preserving functor.

Defining an $I\Sigma$-implementation sentence to be an ordinary $\psi(I\Sigma)$-sentence p we must determine whether an $I\Sigma$-i-model $MA = \langle Ac, \alpha, Aa\rangle$ satisfies p. Since the abstract symbols in $\psi(I\Sigma)$ shall be interpreted by the abstract algebra Aa, the concrete symbols by the concrete algebra Ac, and the abstraction operations by the abstraction function α, we can take the disjoint union of Aa, Ac, and α to obtain a $\psi(I\Sigma)$-algebra interpreting $\psi(I\Sigma)$.

Definition 3.13 [$\text{join}_{I\Sigma}(MA)$]

For an i-signature $I\Sigma = \langle SPa, \sigma, SPc\rangle$ and an $I\Sigma$-i-model $MA = \langle Ac, \alpha, Aa\rangle$

$\text{join}_{I\Sigma}(MA) := Aa \uplus Ac \uplus \alpha$

is the $\psi(I\Sigma)$-algebra A defined by

- for $s \in Sig(SPa)$: $A_s := Aa_s$
- for $s \in Sig(SPc)$: $A_s := Ac_s$
- for $op \in Sig(SPa)$: $op_A := op_{Aa}$
- for $op \in Sig(SPc)$: $op_A := op_{Ac}$
- for $\text{abs-}s \in \text{abs-operations}(I\Sigma)$: $\text{abs-}s_A := \alpha_s$.

The join operator can be extended to a functor from $I\Sigma$-i-models to $\psi(I\Sigma)$-algebras.

Fact 3.14 [$\text{join}_{I\Sigma}$, join]

Defining $\text{join}_{I\Sigma}$ on $I\Sigma$-i-model morphisms $g = \langle hc, ha\rangle$ by

$\text{join}_{I\Sigma}(g) := \{ha_s \mid s \in Sig(SPa)\} \uplus \{hc_s \mid s \in Sig(SPc)\}$

yields a functor

$\text{join}_{I\Sigma}: IMod(I\Sigma) \to EAlg(\psi(I\Sigma))$

and generalizing over all i-signatures yields a natural transformation

$\text{join}: IMod \Longrightarrow EAlg \circ \psi$

3.4 Implementation sentences without hidden components

According to the preceding section we define the set of IΣ-implementation sentences without hidden components or just IΣ-i-sentences to be the set of all ordinary $\psi(I\Sigma)$-sentences. Such an IΣ-i-sentence p is satisfied by an IΣ-i-model MA exactly if MA viewed as the $\psi(I\Sigma)$-algebra $join_{I\Sigma}(MA)$ satisfies p.

Definition 3.15 [ISen1]

The implementation sentence functor without hidden components is given by
$ISen1 := Sen \circ \psi: ISIG \to SET.$

Definition 3.16 [$\models^{i}$]

Let $I\Sigma \in ISIG$, $MA \in IMod(I\Sigma)$ and $p \in ISen1(I\Sigma)$. MA satisfies p, written
$MA \models^{i}_{I\Sigma} p$
iff $join_{I\Sigma}(MA) \models^{e}_{\psi(I\Sigma)} p$.

Fact 3.17 [satisfaction condition]

$\forall\ \tau: I\Sigma_1 \to I\Sigma_2 \in ISIG\ .$
$\forall\ MA \in IMod(I\Sigma_2)\ .$
$\forall\ p \in ISen1(I\Sigma_1)\ .$
$MA \models^{i}_{I\Sigma 2} ISen1(\tau)(p) \iff IMod(\tau)(MA) \models^{i}_{I\Sigma 1} p.$

3.5 The institution

Since the satisfaction condition holds the notions defined above constitute an institution. Like specifications are defined as the theories of the SPEC-institution, implementation specifications will be defined as the theories of this new institution.

Definition 3.18 [IMP1-institution]

IMP1-institution := $\langle ISIG, ISen1, IMod, \models^{i} \rangle$
is the institution of implementation specifications without hidden components.

IMP1 is its category of theories and it is called the category of implementation specifications without hidden components.

Since ISIG is cocomplete, general institution properties tell us that IMP1 is cocomplete as well.

Fact 3.19 [colimits] IMP1 is cocomplete.

3.6 Examples: Implementing sets by lists and lists by array-pointer pairs

In our examples we will assume that the error constants are implicitly declared. As sentences we will use first order formulas where the bound variables are not interpreted as bottom elements. Besides we need some constraint mechanism to exclude unreachable elements (e.g. initial [HKR 80], data [BG 80], hierarchy [SW 82], or algorithmic constraints [BV 85b]).

We will show how several well known implementations of sets by lists can be developed stepwise and hand in hand with the implementing specification.

On the abstract side we have the specification SET of sets with the empty set as constant, and operations to insert an element, to determine or remove the minimum element in a set, and to test for the empty set or for the membership of an element. Beside standard sets, there may be bags or unreachable elements of sort set. The set elements are described in the specification LIN-ORD which introduces a sort elem with an equality operation and an arbitrary reflexive linear ordering. The subspecification BOOL of LIN-ORD specifies the booleans with the usual operations true, false, not, and, or.

On the concrete side the specification LIST extends LIN-ORD to standard lists with the constant nil, the operations cons, car, and cdr, and a test nil? for the empty list. All lists must be generated from the elements by nil and cons. LIST is extended to LIST-S by introducing names for the set simulating operations, but without restricting these operations in order to obtain a variety of different models.

Presentations of the specifications mentioned so far are given in Figure 3.20(a). The sentences parts are not elaborated since the necessary first order formulas are standard and since we did not want to go into the details of the constraint mechanism to be used, because our implementation concept abstracts from these details completely.

We can give a first simple i-specification I:SET/LIST-S from SET to LIST-S:

<u>ispec</u> I:SET/LIST-S =
<u>isig</u> $\sigma_{S/LS}$: SET $\rightarrow$ LIST-S

with the signature morphism

$\sigma_{S/LS}$:	Sig(SET)	$\rightarrow$ Sig(LIST-S)	
	set	$\rightarrow$ list	
	empty	$\rightarrow$ nil	
	empty?	$\rightarrow$ nil?	
	insert	$\rightarrow$ l-insert	
	in?	$\rightarrow$ l-in?	
	min	$\rightarrow$ l-min	
	remove-min	$\rightarrow$ l-remove-min	
	x	$\rightarrow$ x	for x ε Sig(LIN-ORD)

It merely defines the signature morphism $\sigma_{S/LS}$ translating sort set to list and translating the set operations to their simulating list operations without renaming the signatures of the common subspecifications LIN-ORD and BOOL. Since I:SET/LIST-S contains no i-sentences, its i-models comprise all possible representations of sets by lists.

I:SET/LIST-S can be refined in various ways by adding i-sentences restricting the abstraction operations of sort set, such that e.g.

- <u>a</u>ll lists represent sets (IA:SET/LIST-S),
- only lists with <u>u</u>nique entries may represent sets (IU:SET/LIST-S),
- only <u>s</u>orted lists may represent sets (IS:SET/LIST-S), or
- only <u>s</u>orted lists with <u>u</u>nique entries may represent sets (ISU:SET/LIST-S).

The last i-specification refines not only I:SET/LIST-S, but also IU:SET/LIST-S and IS:SET/LIST-S. The i-specifications are given in Figure 3.21 where we use abs-s: $\sigma_{S/LS}(s) \rightarrow s$ as the abstraction operation name of sort s.

Corresponding to the four alternative refinements of I:SET/LIST-S we could now

specify alternative refinements of the concrete LIST-S specification by adding sentences fixing the set simulating operations. The resulting LIST-S refinements could in turn be used to refine the respective i-specifications by replacing LIST-S by the corresponding LIST-S refinement.
Here, however, we want to carry out the development in another direction by implementing the lists by array-pointer pairs. For this purpose we consider the three specifications listed in Figure 3.20(b): PAIR introduces standard arrays and pairs of an array with a natural number. PAIR-L fixes the new LIST simulating operations such that p-nil yields the new array with pointer zero, p-nil? checks whether the pointer is zero, p-cons puts an element in the field indicated by the pointer and increments the pointer by one, p-car gets the element in the field indicated by the pointer minus one, and p-cdr decrements the pointer by one. In contrast, the new operations in PAIR-LS are unrestricted in order to allow for a variety of i-models differing in their SET simulating operations.

In the i-specification implementing LIST by PAIR-L given by

```
i-spec  I:LIST/PAIR-L  =
    isig  σ_L/LP: LIST → PAIR-L
    isentences
         ∀ p: pairs .
             (p-nil?(p) = true  => abs-list(p) = nil)                       &
             (p-nil?(p) = false =>
                abs-list(p) = cons(abs-elem(p-car(p)),abs-list(p-cdr(p))) )
```

with the signature morphism

$\sigma_{L/PL}$: Sig(LIST) → Sig(PAIR-L)

list	→ pairs	
op	→ p-op	for op ε {nil, nil?, cons, car, cdr}
x	→ x	otherwise

the abstraction operation abs-list is fixed. This i-specification can be extended to an i-specification implementing LIST-S by PAIR-LS:

```
ispec  I:LIST-S/PAIR-LS  =  I:LIST/PAIR-L  u
    isig  σ_LS/PLS: LIST-S → PAIR-LS
```

with the signature morphism

$\sigma_{LS/PLS}$: Sig(LIST-S) → Sig(PAIR-LS)

l-op	→ p-op	for op ε {insert ,in? ,min ,remove-min}
x	→ $\sigma_{L/PL}(x)$	otherwise

comprising all i-models with fixed LIST simulating operations but with varying SET simulating operations. Figure 3.22 shows the relations between the specifications and i-specifications developed so far.

4. Implementation specifications with hidden components

The inclusion of hidden specification parts into an ADT specification technique usually extends its expressive power ([TWW 82], [BBTW 81]), and when describing the composition of algebraic implementations hidden components are needed for the intermediate specification part ([EKMP 82]).
We will now extend the IMP1-institution by so-called hidden specification sentences which are comparable to an algebraic specification mechanism called functor image restriction in [Ehg 81], reflections in [EWT 82] and derive ... from ... by-construct

```
   sorts elem
   ops   eq, le: elem elem → bool
   sentences ... < specifying eq as
        equality and le as an arbitrary
        reflexive linear ordering >

spec SET = LIN-ORD u
   sorts set
   ops   empty: → set
         insert: elem set → set
         min: set → elem
         remove-min: set → set
         empty?: set → bool
         in?: elem set → bool
   sentences ... < specifying the set
        operations with their usual
        meaning, but not necessarily
        excluding non-standard sets >

spec LIST = LIN-ORD u
   sorts list
   ops   nil: → list
         cons: elem list → list
         car: list → elem
         cdr: list → list
         nil?: list → bool
   sentences ... < specifying standard
        lists over elem generated by
        nil and cons >

spec LIST-S = LIST u
   ops   l-insert: elem list → list
         l-min: list → list
         l-remove-min: list → list
         l-in?: elem list → bool
```

```
spec PAIR = LIN-ORD u NAT u
   sorts  array, pairs
   ops    new: → array
          put: array nat elem → array
          get: array nat → elem
          pair: array nat → pairs
          pa: pairs → array
          pn: pairs → nat
   sentences ...<specifying standard
        arrays and pairs of an array
        with a natural number>

spec PAIR-L = PAIR u
   ops p-nil: → pairs
       p-nil?: pairs → bool
       p-cons: elem pairs → pairs
       p-car: pairs → elem
       p-cdr: pairs → pairs
   sentences ...<specifying the
       LIST-simulating such that
       p-cons puts an elementinto
       the array and increments the
       pointer by one, p-cdr
       decrements the pointer by one,
       etc.>

spec PAIR-LS = PAIR-L
   ops p-insert: elem pairs → pairs
       p-in?: elem pairs → bool
       p-min: pairs → elem
       p-remove-min: pairs → pairs
```

Figure 3.20 The ADT specifications in the implementations of sets by lists (a) and of lists by array-pointer pairs (b)

```
ispec IA:SET/LIST-S = I:SET/LIST-S u
    isentences
       (∀ x: list . ∀ e: elem.
            abs-set(cons(e,x)) = insert(abs-elem(e),abs-set(x)))

ispec IS:SET/LIST-S = I:SET/LIST-S u
    isentences
       (∀ e, e1, e2:elem . ∀ x: list .
            abs-set(cons(e,nil)) = insert(abs-elem(e),empty) &
            le(e1,e2) = true & eq(e1,e2) = false =>
               abs-set(cons(e1,cons(e2,x))) =
                   insert(abs-elem(e1),abs-set(cons(e2,x)))  &
            le(e2,e1) = true & eq(e1,e2) = false =>
               abs-set(cons(e1,cons(e2,x))) = error-set        )

ispec IU:SET/LIST-S = I:SET/LIST-S u
    isentences
       (∀ e, e1, e2: elem . ∀ x: list .
            abs-set(cons(e,nil)) = insert(abs-elem(e),empty) &
            (in?(e,abs-set(x)) = true =>
                abs-set(cons(e,x)) = error-set)                )

ispec ISU:SET/LIST-S = IU:SET/LIST-S u IS:SET/LIST-S
```

Figure 3.21 Some i-specifications implementing sets by lists

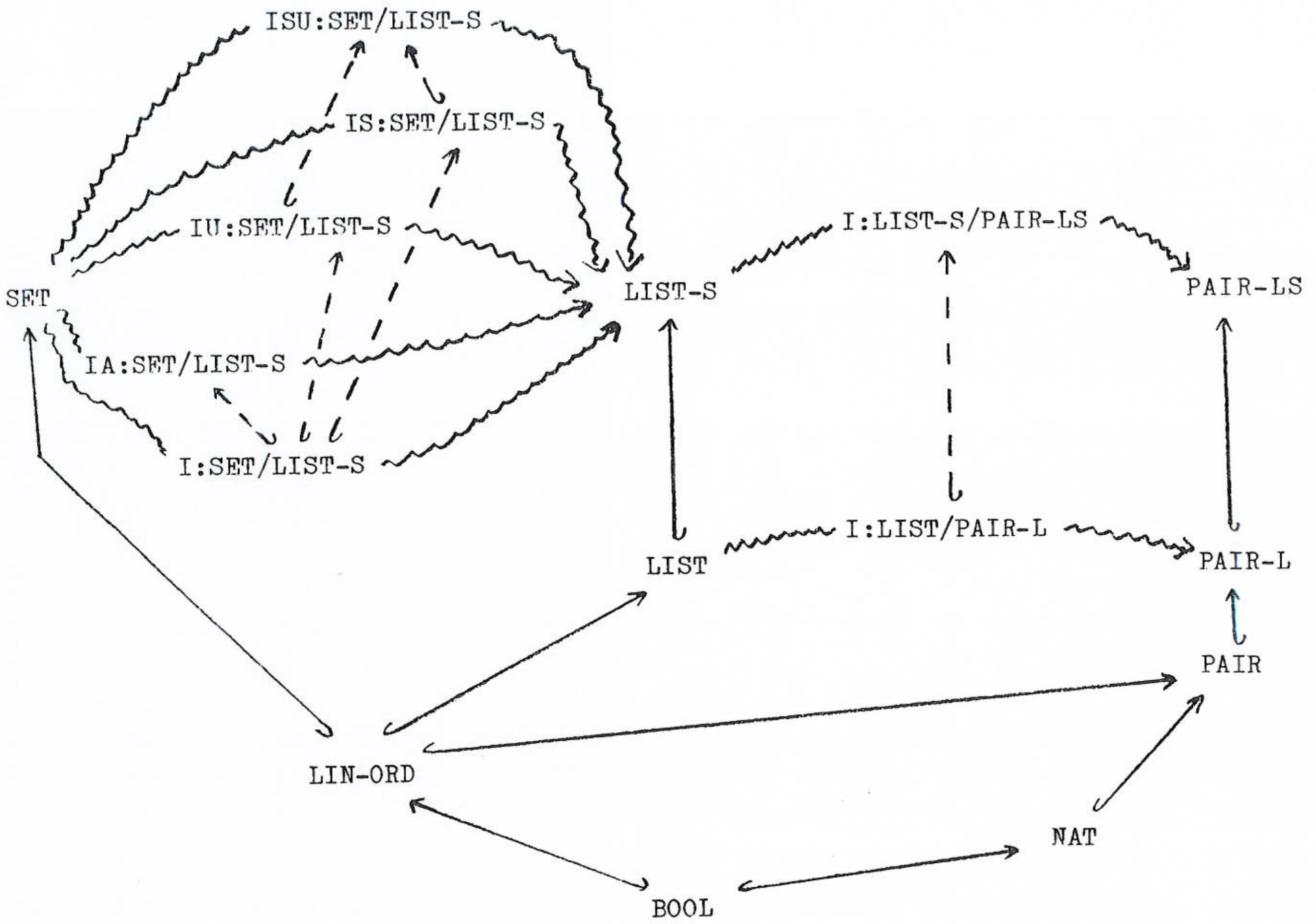


Figure 3.22: The relation between the specifications and i-specifications

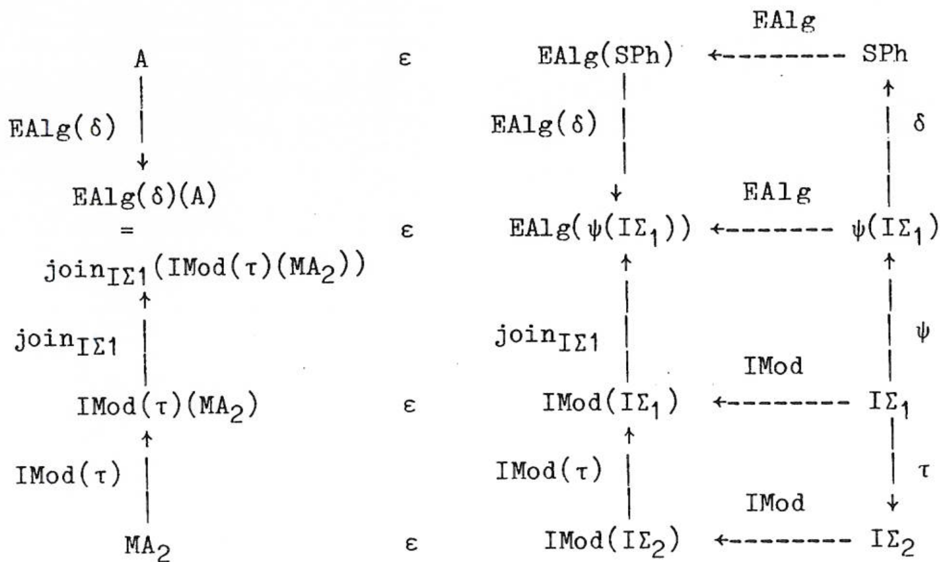


Figure 4.2 Satisfaction of a hidden specification sentence
$\langle SPh,\ \delta: \psi(I\Sigma_1) \to SPh,\ \tau: I\Sigma_1 \to I\Sigma_2 \rangle$

in [SW 83]. Wheras so far an implementation sentence over an i-signature $I\Sigma$ is an ordinary sentence over the vocabulary $\psi(I\Sigma)$, we now extend $\psi(I\Sigma)$ by arbitrary hidden sorts and operation symbols from an ordinary ADT specification SPh via some signature morphism $\delta: \psi(I\Sigma) \rightarrow Sig(SPh)$.

Definition 4.1 [hidden specification sentence, HSen]

Let SPh ε SPEC, $\delta \varepsilon$ /SIG/ and $\tau \varepsilon$ /ISIG/. An $I\Sigma_2$-hidden specification sentence is a triple

$$sh = <SPh, \delta: \psi(I\Sigma_1) \rightarrow Sig(SPh), \tau: I\Sigma_1 \rightarrow I\Sigma_2>$$

and $HSen(I\Sigma_2)$ denotes the set of all $I\Sigma_2$-hidden specification sentences. The translation of sh by $\tau': I\Sigma_2 \rightarrow I\Sigma'$ is the $I\Sigma'$-hidden specification sentence given by $HSen(\tau')(sh) := <SPh, \delta, \tau' o \tau>$

Note that the third component τ of a hidden specification sentence allows for the translation of such sentences by arbitrary (i-signature) morphisms, thus surving the same function as the second component of the data constraints in [GB 83].

Writing sh for an $I\Sigma_2$-i-specification we are interested only in those $I\Sigma_2$-i-models MA that can be `extended' to SPh models: MA satisfies sh iff MA forgotten along τ and viewed as a $\psi(I\Sigma)$-algebra is identical to some SPh-algebra A where the hidden part of A is forgotten along δ. This situation is illustrated in Figure 4.2 and made precise as follows:

Definition 4.3 [satisfaction $\models^i$ of a hidden specification sentence]

$$\forall MA \; \varepsilon \; IMod(I\Sigma_2) \; . \; MA_2 \models^i <SPh, \delta, \tau> \;\; <=> \;\; \exists A \; \varepsilon \; EAlg(SPh) \; . \; EAlg(\delta)(A) = join_{I\Sigma 1}(IMod(\tau)(MA))$$

Example 4.4 [implementing sets via lists by array-pointer pairs]

An i-specification describing the composition of ISU:SET/LIST-S and I:LIST-S/PAIR-LS is

ispec ISU:SET/PAIR-LS
  isig $\sigma_{LS/PLS} \circ \Sigma_{S/LS}$: SET $\rightarrow$ PAIR-LS
  hidden-spec-sentences <S-LS-PLS, δ, id>

with the hidden specification
spec S-LS-PLS = SET u LIST-S u PAIR-LS u
  ops $abs\text{-}set_{S/LS}$: list $\rightarrow$ set
    $abs\text{-}list_{LS/PLS}$: pairs $\rightarrow$ list
    $abs\text{-}set_{S/PLS}$: pairs $\rightarrow$ set
  sentences
    i-sentences(ISU:SET/LIST-S) u
    i-sentences(I:LIST-S/PAIR-LS) u
    ($\forall$ p: pairs .
      $abs\text{-}set_{S/PLS}(p) = abs\text{-}set_{S/LS}(abs\text{-}list_{LS/PLS}(p))$)

where the abstraction operations of the individual implementations are indexed correspondingly and where δ maps abs-set to $abs\text{-}set_{S/PLS}$. Note that for every denoted i-model $<A_{PAIR-LS}, \alpha, A_{SET}>$ there must exist a LIST-S model A_{LIST-S} and abstraction functions α_1 and α_2 such that $<A_{LIST-S}, \alpha_1, A_{SET}>$ is an i-model of ISU:SET/LIST-S, <APAIR-LS,α2,ALIST-S> is an i-model of I:LIST-S/PAIR-LS, and α is the composition of α1 and α2.

Fact 4.5 ⌊satisfaction condition for hidden specification sentences⌋

$$\forall \tau': I\Sigma_2 \rightarrow I\Sigma' \in ISIG \;.\; \forall MA \in IMod(I\Sigma') \;.\; \forall sh \in HSen(I\Sigma_2) \;.\; MA \models^{i}_{I\Sigma'} HSen(\tau')(sh) \iff IMod(\tau')(MA) \models^{i}_{I\Sigma 2} sh.$$

Definition 4.6 ⌊IMP-institution, IMP⌋

IMP-institution := $<ISIG, ISen, IMod, \models^{i}>$
is the institution extending the IMP1-institution by $ISen(I\Sigma) := ISen1(I\Sigma) \cup HSen(I\Sigma)$. IMP is the category of theories of this institution and called the category of implementation specifications (with hidden components).

5. Normal forms

The introduction of hidden specification sentences allows us to derive a very useful normal form result:

Fact 5.1 ⌊normal form⌋

Any i-specification ISP $\in$ IMP can be transformed into an equivalent i-specification ISP′ in normal form having exactly one hidden specification sentence.

Proof: (idea) Given two hidden specification sentences

$$sh_j = <SP_j, \rho_j: \psi(I\Sigma_j) \rightarrow Sig(SP_j), \tau_j: I\Sigma_j \rightarrow I\Sigma>$$

we can merge sh_1 and sh_2 by taking the coproduct SP_{cop} of SP_1 and SP_2, taking the coproduct $I\Sigma_{cop}$ of $I\Sigma_1$ and $I\Sigma_2$, and the uniquely determined morphisms ρ' and τ' as given in

$$sh = <SP_{cop}, \rho': \psi(I\Sigma_{cop}) \rightarrow Sig(Sp_{cop}), \tau': I\Sigma_{cop} \rightarrow I\Sigma>$$

To show that

$$IMod(I\Sigma, \{sh_1, sh_2\}) = IMod(I\Sigma, \{sh\})$$

and the generalization of this statement to (possibly infinite) sets of sentences relies on the fact that the model functor EAlg of the underlying SPEC-institution respects coproducts.

The i-specification ISU:SET/PAIR-LS of Example 4.4 is in normal form.

6. Composition

6.1 Composition of implementation signatures

The composition of i-signatures is given by the composition of their signature translations.

Definition 6.1

For $I\Sigma_j = <SPa_j, \sigma_j, SPc_j>$ with $j \in \{1,2\}$ and $SPc_2 = SPa_1$ the composition of $I\Sigma_1$ and $I\Sigma_2$ is given by

$$I\Sigma_1 \bullet I\Sigma_2 := <SPa_1, \sigma_2 \circ \sigma_1, SPc_2>$$

Obviously, this composition operation is associative.

6.2 Composition of implementation models

Having already defined the composition of i-signatures the composition of an $I\Sigma_2$-i-

model $MA_2 = <Ac_2, \alpha_2, Aa_2>$ with an $I\Sigma_1$-i-model $MA_1 = <Ac_1, \alpha_1, Aa_1>$ where $Aa_2 = Ac_1$ should yield an $I\Sigma_1 \bullet I\Sigma_2$-i-model resulting from the composition of their abstraction functions. However, in order to be able to compose the abstraction functions we must first apply the forgetful functor $PEAlg(\sigma_1)$ to α_2:

Fact 6.2 [composition of implementation models]

(1) The composition of MA_2 and MA_1 as given by

$$MA_2 \bullet MA_1 := <Ac_2, \alpha_1 \circ PEAlg(\rho_1)(\alpha_2), Aa_1>$$

is an $I\Sigma_1 \bullet I\Sigma_2$-i-model

(2) The composition operation on i-models is associative.

Given two i-specifications ISP_1 and ISP_2 with composable i-signatures, we can now compose all their respective i-models and obtain a subcategory of $IMod(I\Sigma_1 \bullet I\Sigma_2)$.

Definition 6.3 [composition of i-specification model categories]

$$IMod(ISP_2) \bullet IMod(ISP_1)$$

is the full subcategory of $IMod(I\Sigma_1 \bullet I\Sigma_2)$ generated by all $MA_2 \bullet MA_1$ with $MA_j \in IMod(ISP_j)$.

Fact 6.4 The composition operation on i-model categories is associative.

6.3 Composition of implementation specifications

In Section 2 we already motivated our requirement that the compositions of i-specifications and their i-model categories should be compatible. With the notions introduced above we can formalize this compatibility condition by requiring

$$IMod(ISP_1 \bullet ISP_2) = IMod(ISP_2) \bullet IMod(ISP_1)$$

At least two questions arise immediately: Does there exists an i-specification $ISP = ISP_1 \bullet ISP_2$ describing the composition of ISP_1 and ISP_2 such that the compatibility condition is satisfied? And secondly, is there a constructive way to generate ISP for given ISP_1 and ISP_2? The following fact answers both questions in the affirmative.

Fact 6.5 [composition of i-specifications]

For any two i-specifications ISP_1 and ISP_2 with composable i-signatures there exists an i-specification

$$ISP_1 \bullet ISP_2$$

such that the compatibility condition is satisfied.

Proof: (idea) In order to generalize the construction carried out in Example 4.4 we construct normal form presentations $ISPn_1$ and $ISPn_2$ which exist according to Fact 5.1, and perform the following steps:

(1) Combine the hidden specifications SPh_1 and SPh_2 of $ISPn_1$ such that the middle specification $SPa_2 = SPc_1$ is identified in this combination.

(2) Add the abstraction operations of $I\Sigma_1$, $I\Sigma_2$, and $I\Sigma_1 \bullet I\Sigma_2$, and add the corresponding composition axioms.

Steps (1) and (2) yield a hidden specification SPh describing a normal form representation of $ISP_1 \bullet ISP_2$.

As an illustration of Fact 6.5 consider again the composed i-specification ISU:SET/PAIR-LS from Example 4.4 which was constructed correspondingly yielding ISU:SET/LIST-S $\bullet$ I:LIST-S/PAIR-LS: The intermediate LIST-S specification is contained

in the hidden specification, and the i-models denoted by the composed implementation are exactly those models that can be composed from the models of the two individual implementations. Similarly, we could compose I:LIST-S/PAIR-LS with any of the other four set-by-list implementations (c.f. Figure 3.22), yielding four different implementations of sets by array-pointer pairs.

Fact 6.6 The composition operation on i-specifications is associative.

Proof: By Fact 6.4 since the compatability condition holds.

7. Implementation specifications in the ISDV system

As already pointed out in Sections 1 and 2, there is a close correspondence between loose specifications and i-specifications w.r.t. their role in software development. Loose specifications provide a means for a formalized stepwise refinement scenario and the same is true for our implementation concept. The addition of new constraints to a loose specification corresponds to making further design decisions; likewise, the addition of i-sentences to an i-specification corresponds to further design decisions influencing e.g. the efficiency of certain operations (c.f. the set-by-list example in Section 3.6). Care must be taken because the process of refinement may yield an inconsistent specification having no models any more, and the same may happen to i-specifications. For both situations the same techniques can be used to cope with this problem, e.g. suitably restricting the class of admissible sentences or providing a constructively defined model. Such a model may be the program obtained gradually during the development process.

The latter approach is supported in the specification development language ASPIK within the Integrated Software Development and Verification (ISDV) system ([BV85b], [BOV 86]). ASPIK provides a uniform integration of high level axiomatic and lower level constructive specification techniques. An essential part of specification development in ASPIK is the gradual refinement of axiomatic parts by constructively defined models. That means an initially completely axiomatic specification is guaranteed to be consistent if its refinement process can be carried through to a completely constructive specification. This approach provides also a means for coping with the consistency problem of composed i-specifications: In general a composed i-specification may have no models although its component i-specifications have models. This cannot happen if the intermediate specification is refined to a model that lies in both classes of i-models.

Beside consistency, the SW-property of i-specifications is another model theoretically defined notion: According to the approach of Sannella and Wirsing in [SW 82] we call an i-specification ISP = $\langle I\Sigma, IE\rangle$ with $I\Sigma = \langle SPa, \sigma, SPc\rangle$ an SW-implementation if IE is empty and if for every SPc-model Ac there exists an $I\Sigma$-i-model $\langle Ac, \alpha, Aa\rangle$. This definition shows that in the approach of [SW 82] syntax and semantics of implementations are not clearly distinguished. Based on the work of [SW 82], Urbassek [Urb 85] has developed syntactic criteria for i-specifications in ASPIK that guarantee the SW-property. While these criteria can also be used to guarantee the consistency of composed i-specifications, less restrictive syntactic criteria should be developed that relax the SW-property so that not every concrete model must implement an abstract one.

8. Conclusions

Our implementation concept for loose abstract data type specifications completely distinguishes between the syntactical level of specifications and the semantical level of models by introducing the notions of implementation signatures, - models,

and - specifications. It provides the notion of implementation refinement which is not present in other approaches. An implementation in the approach for Clear-like specifications proposed in [SW 82] is - in our terminology - an i-signature with the semantic condition that for every abstract algebra there is a concrete one with an abstraction function in between. Concepts like those of [GM 82] and [Sch 82] are based on behavioural abstraction and have been proposed for modules, and [Hup 80] considers implementations between canon specifications. The implementation concept for the kernel language ASL of [SW 83] merely requires that the abstract specification is included in the concrete one. This simple notion is based on the fact that, as a semantical language, ASL has very powerful specification building operations which however may not be present in a language for ADT specifications.

In e.g. [GM 82], [SW 82], [SW 83] and in the initial approach of [EKMP 82] implementation composition is defined and is explicitly shown to be associative. Whereas in the former composition is a totally defined operation this is true in [EKMP 82] for so-called weak implementations and for a particular class of strong implementations. Of the cited approaches only [EKMP 82] distinguishes completely between syntactical and semantical levels which is a prerequisite for studying the compatability problem of a composition operation. However, this problem is not addressed explicitly since every specification denotes a unique algebra and no explicit definition of a semantical composition operation is given.

Whereas the composition discussed in this paper is usually called vertical there is also a horizontal composition arising in the context of parameterized specifications (see e.g. [EK 82], [GM 82], [SW 82]). For the implementation concept proposed here we show in [BV 85b] that horizontal composition and instantiation of parameterized implementations are compatible with vertical composition, allowing to combine implementation specifications interchangeably in different directions with the same result.

References

[BBTW 81] Bergstra, J.A., Broy, M., Tucker, J.V., Wirsing, M.: On the power of algebraic specifications. Proc. 10th MFCS, Strbske Pleso, Czechoslovakia. LNCS Vol. 118, pp. 193-204, 1981.

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

[BOV 86] Beierle,C., Olthoff, W., Voß, A.: Towards a formalization of the software development process. Proc. Software Engineering '86, Southampton, 1986

[BV 85a] Beierle, C., Voß, A.: Implementation specifications. In: H.-J. Kreowski (ed): Recent Trends in Data Type specifications. Informatik Fachberichte 116, Springer Verlag, 1985.

[BV 85b] Beierle, C., Voß, A.: Algebraic specifications and implementations in an integrated software development and verfication system. Memo SEKI-85-12, FB Informatik, Univ. Kaiserslautern (joint SEKI-Memo containing the Ph.D. thesis by Ch. Beierle and the Ph.D. thesis by A. Voß), Dec. 1985

[Ehc 82] Ehrich, H.-D.: On the theory of specification, Implementation and Parametrization of Abstract Data Types. JACM Vol. 29, No. 1, Jan. 1982, pp. 206-227.

[Ehg 81] Ehrig, H.: Algebraic Theory of Parameterized Specifications with Requirements. Proc. 6th Colloquium on Trees in Algebra and Programming (E. Astesiano, C. Böhm, eds.), LNCS 112, pp. 1-24, 1981.

[EKMP 82] Ehrig, H., Kreowski, H.-J., Mahr, B., Padawitz, P.: Algebraic Implementation of Abstract Data Types. Theor. Computer Science Vol. 20, 1982, pp. 209-254.

[EKP 78] Ehrig, H.,Kreowski, H.J.,Padawitz, P.: Stepwise specification and implementation of abstract data types. Proc. 5th ICALP, LNCS Vol. 62, 1978, pp. 203-206.

[EWT 82] Ehrig, H., Wagner, E., Thatcher, J.: Algebraic Constraints for specifications and canonical form results. Draft version, TU Berlin, June 1982.

[Ga 83] Ganzinger, H.: Parameterized Specifications: Parameter Passing and Implementation with respect to Observability. ACM TOPLAS Vol. 5, No.3, July 1983, pp. 318-354.

[GB 83] Goguen, J.A., Burstall, R.M.: Institutions: Abstract Model Theory for Program Specification. Draft version. SRI International and University of Edinburgh, January 1983.

[GM 82] Goguen, J.A., Meseguer, J.: Universal Realization, Persistent Interconnection and Implementation of Abstract Modules. Proc. 9th ICALP, LNCS 140, 1982, pp. 265-281.

[GTW 78] Goguen, J.A., Thatcher, J.W., Wagner, E.G.: An initial algebra approach to the specification, correctness, and implementation of abstract data types, in: Current Trends in Programming Methodology, Vol.4, Data Structuring (ed. R. Yeh), Prentice-Hall, 1978, pp. 80-144.

[Hup 80] Hupbach, U.L.: Abstract implementation of abstract data types. Proc. 9th MFLS, Rydzyna, Poland. LNCS, Vol. 88, pp. 291-304, 1980.

[HKR 80] Hupbach, U.L., Kaphengst, H., Reichel, H.: Initial algebraic specifications of data types, parameterized data types, and algorithms. VEB Robotron, Zentrum für Forschung und Technik, Dresden, 1980.

[Li 83] Lipeck, U.: Ein algebraischer Kalkül für einen strukturierten Entwurf von Datenabstraktionen. Dissertation. Forschungsbericht Nr. 148, Universität Dortmund, 1983.

[Sch 82] Schoett, O.: A theory of program modules, their specification and implementation. Draft report, Univ. of Edinburgh.

[SW 82] Sannella, D.T., Wirsing, M.: Implementation of parameterized specifications, Proc. 9th ICALP 1982, LNCS Vol. 140, pp 473 - 488.

[SW 83] Sannella, D., Wirsing, M.: A kernel language for algebraic specification and implementation. Proc. FCT, LNCS Vol. 158, 1983.

[TWW 82] Thatcher, J.W., Wagner, E.G., Wright, J.B.: Data Type Specification: Parameterization and the Power of Specification Techniques. ACM TOPLAS Vol. 4, No. 4, Oct. 1982, pp. 711-732.

[Urb 85] Urbassek, C.: Ein Implementierungskonzept für ASPIK-Spezifikationen und Korrektheitskriterien. Diploma thesis, Univ. Kaiserslautern, 1985.