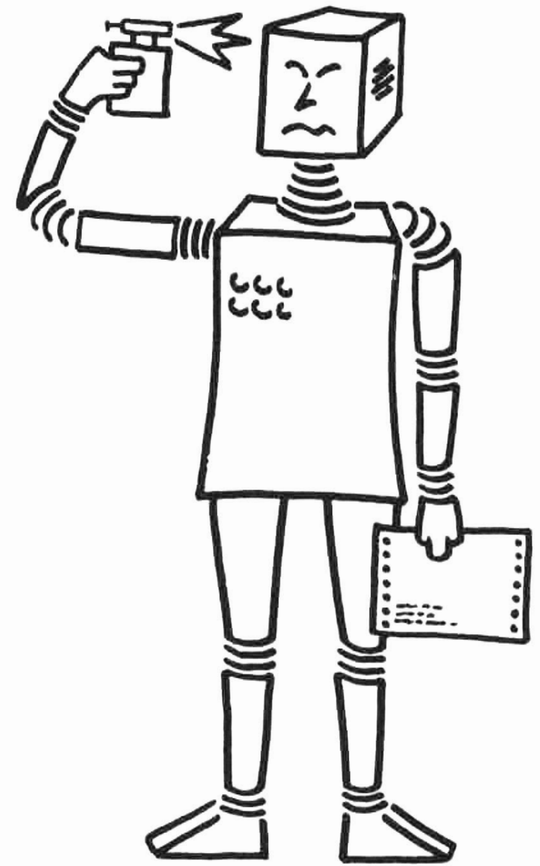
SEKI-PROJEKT

SEKI MEMO

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



Implementation Specifications

Ch. Beierle, A. Voß

MEMO SEKI-85-08

IMPLEMENTATION SPECIFICATIONS *

Christoph Beierle, Angelika Voß
Fachbereich Informatik, Universität Kaiserslautern
Postfach 3049, 6750 Kaiserslautern, West Germany

Abstract

Loose specifications of abstract data types (ADTs) have many non-isomorphic algebras as models. An implementation between two loose specifications should therefore consider many abstraction functions together with their source and target algebras. Just like specifications are stepwise refined to restrict their class of models, implementations should be stepwise refinable to restrict the class of abstraction functions. In this scenario specifications and implementations can be developed interwovenly.
For example, we can consider implementations of sets by lists where the set simulating list operations are still left open. They may be refined later on so that an implementation of sets by arbitrary lists, by lists without double entries, or by sorted lists is obtained, differing e.g. in the efficiency of the set simulating operations.
We suggest to have implementation specifications analogously to loose ADT specifications: Implementations have signatures, models, axioms and sentences thus constituting an institution. Implementation specifications are the theories of this institution and refinements between implementation specifications are its theory morphisms.

1. Introduction

Fixed ADT specifications with only isomorphic models were studied before loose ADT specifications with non-isomorphic models, and several implementation concepts have been proposed, discussed, and revised for fixed specifications (e.g. [GTW 78], [Ehc 82], [EKP 78], [EKMP 82], [Ga 83]). By now there seems to be a basic consent that such an implementation concept should incorporate the following notions:

- an abstract specification to be implemented,
- a concrete specification implementing the abstract one,
- a signature morphism from the abstract to the (possibly extended) concrete specification allowing to translate abstract terms to concrete ones, and
- an abstraction function from the concrete to the abstract algebra allowing to translate the concrete value of a concrete term back to an abstract value. Abstraction functions need not be totally defined, but must be surjective and homomorphic w.r.t. their domain of definition.

In contrast, so far only one implementation concept has been proposed for loose specifications that generalizes the fixed case, namely the concept proposed by Sannella and Wirsing in [SW 82]. Our own implementation concept generalizes that of Sannella and Wirsing giving room to a refinement process between implementations. Moreover, our approach abstracts from a particular ADT specification method by using the notion of an institution ([GB 83]) which provides abstract characterizations of signatures, models, sentences etc.

In Section 2 we outline the basic idea of our implementation concept. In Section 3 we briefly state the assumptions about the underlying loose specifications which are fundamental for our development. In Section 4 we introduce the institution of implementation specifications. In Section 5 we illustrate how implementations of sets by lists can be developed and refined stepwise hand in hand with the loose specifications. Section 6 contains a summary and a comparison.

* to appear in: Proc. of the 3rd Workshop on Theory and Applications of Abstract Data Types, Informatik Fachberichte, Springer Verlag (1985)

2. Basic idea

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.

Having translated an abstract term into a concrete one we are faced with the following questions:

(1) In which concrete algebra shall the concrete term be evaluated, since there may be many non-isomorphic algebras?

(2) To which abstract algebra shall we translate the value of the concrete term, since there also may be many non-isomorphic algebras?

(3) Which abstraction function shall be used for the translation, since there may be different ones?

Having answered these questions we may further ask:

(4) How can we specify the selected concrete algebras, abstract algebras, and abstraction functions in an 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.

Since the idea of loose specifications is to consider at first an arbitrary 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 and answering (1) - (3) we introduce the notion of implementation models:

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

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. Thus we solve problem (4) w.r.t. the algebras by allowing sentences over the abstract and the concrete signature to be given in an implementation.

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 we solve problem (4) completely by admitting arbitrary sentences over the abstract and the concrete signatures extended by the abstraction operation names. These sentences will be called implementation sentences.

Summarizing we propose an implementation specification to be

- a simple implementation
- together with a set of implementation sentences 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 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.

Since an implementation contains two specifications, an implementation morphism should consist of two specification morphisms, an abstract one between the abstract specifications and a concrete one between the concrete specifications. With these two morphisms, the sentences of the refined implementation can be translated into sentences over the refining implementation by mapping the sorts and operations according to the two specification morphisms and by mapping the abstraction operation names to the corresponding abstraction operation names in the refining implementation.

Thus, a refinement between two implementations is given by an abstract and a concrete specification morphism such that the translated sentences of the refined implementation hold in the refining one.

3. 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]).

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

is an institution where

- SIG is a category of equational signatures with an error constant error-s for each sort s.

- EAlg is a 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.

- $|\overset{e}{=}$ 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.

4. The institution of implementation specifications

In order to develop our implementation concept in the framework of institutions we will have to make precise the notions of

- implementation signatures (Section 4.1) and
- implementation models (Section 4.2).

Having determined these notions we will establish a connection between loose specifications and implementations in Section 4.3 which will be helpful to formalize in Section 4.4 the remaining notions of

- implementation sentences and
- satisfaction of an implementation sentence by an implementation model.

Section 4.5 contains a summary of the new institution.

4.1 Implementation signatures

The signature Σ of a loose specification SP = $\langle\Sigma,E\rangle$ may be viewed as a simple specification which has no sentences at all:

$$\Sigma \cong \langle\Sigma,\emptyset\rangle.$$

This suggested to define an implementation signature to be a simple implementation specification which has no sentences.

According to Section 2, such a simple implementation consists of an abstract specification SPa = $\langle\Sigma a,Ea\rangle$, a concrete specification SPc = $\langle\Sigma c,Ec\rangle$, and a signature morphism σ: $\Sigma a \to \Sigma c$ translating the abstract to the concrete signature. Thus, an implementation signature, or shorter i-signature IΣ is a triple

$$I\Sigma = \langle SPa,\sigma,SPc\rangle.$$

We already suggested that a refinement between two implementations should consist of two specification morphisms between the abstract specifications and between the concrete specifications. Since an implementation comprises in particular an i-signature, we obtain the notion of refinement or morphism between i-signatures: An i-signature morphism

$$\tau: I\Sigma_1 \to I\Sigma 2$$

between two i-signatures $I\Sigma_j = \langle SPa_j,\sigma_j,SPc_j\rangle$ for $j \in \{1,2\}$ is a pair

$$\tau = \langle\rho a,\rho c\rangle$$

consisting of an abstract specification morphism ρa: $SPa_1 \to SPa_2$ and a concrete specification morphism ρc: $SPc1 \to SPc_2$. The refinement requirement that the translated sentences of $I\Sigma 1$ must hold in $I\Sigma 2$ is trivially satisfied since $I\Sigma 1$ has no sentences at all.

However, another requirement should also be satisfied: Assume we have an i-signature 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 that means that the specification morphisms ρa and ρc should be compatible with the signature morphisms σ_1 and σ_2. This in turn 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.

These notions of i-signatures and i-signature morphisms constitute a category. In fact, it is the comma category induced by the functor Sig forgetting specifications to their signatures.

Definition 4.1 [ISIG, i-signature]

Given the forgetful functor Sig: SPEC → Sig, the comma category
ISIG = (Sig↓Sig)
is the category of implementation signatures (i-signatures).

For an i-signature $I\Sigma = \langle SPa, \sigma, SPc\rangle \in ISIG$, SPa is called the abstract specificatation of IΣ, SPc the concrete specification, and σ the translation. For an i-signature morphism $\tau = \langle \rho a, \rho c\rangle: I\Sigma_1 \to I\Sigma_2$, ρa is called the abstract specification morphism and ρc the concrete one.

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 4.2 [colimits]

ISIG is cocomplete.

4.2 Implementation models

In Section 2 we already discussed the meaning of a simple implementation. Thus, for an i-signature IΣ an IΣ-implementation model should consist of a concrete SPc-algebra Ac, an abstract SPa-algebra Aa, and a partial, homomorphic, surjective abstraction function α from Ac to Aa. More precisely, for $\Sigma a = \langle Sa, Opa\rangle$ α is an Sa-indexed family of functions α_s, each going into an abstract carrier Aa_s and starting from the corresponding concrete carrier $Ac_{\sigma(s)}$.

As in the fixed case, the concrete carriers and operations which are not needed for the translation can be forgotten along σ so that we obtain FAlg(σ)(Ac), i.e. Ac viewed as a Σa-algebra, as the source of the abstraction function α.

Now we could define α: EAlg(σ)(Ac) → Aa as a partial, surjective homomorphism. However, 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 need to hold only if $\alpha(x)$ is non-error.

Calling a family of functions partially homomorphic if it is homomorphic except for the error elements, we can define an abstraction function

$$\alpha: EAlg(\sigma)(Ac) \rightarrow Aa$$

to be a surjective, partially homomorphic family of functions. Thus an $I\Sigma$-implementation model, or just $I\Sigma$-i-model MA is a tripel

$$MA = \langle Ac, \alpha, Aa \rangle.$$

Note that in contrast to $I\Sigma$, where the first component is the abstract one and the third is the concrete one, we now have the abstract algebra in the third component and the concrete algebra in the first component. However, in both cases the first component contains the source and the third component the target of the function in the middle component.

Proceeding analogously to i-signature morphisms, we obtain a notion of i-model morphisms as a connection between two i-models. Given another $I\Sigma$-i-model

$$MB = \langle Bc, \beta, Ba \rangle$$

an i-model morphism from MA to MB should consist of

- an SPa-homomorphism $ha: Aa \rightarrow Ba$ and
- an SPc-homomorphism $hc: Ac \rightarrow Bc$.

Analogously to i-signature morphisms the compatibility condition for i-model morphisms should express that 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. This condition may be expressed graphically by requiring that the square

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

commutes. Note that we must forget hc along σ because we also forget its source and target along σ.

To formalize this description we first have to solve a technical problem: In order to require that the square above commutes the morphisms occurring in there must belong to the same category. However, hc and ha are homomorphisms while α and β as abstraction functions are functions which are surjective but which are only partially homomorphic in general.

Since every homomorphism is partially homomorphic, but is not necessarily surjective the appropriate category for the square should have strict algebras as objects and partially homomorphic functions as homomorphisms.

<u>Definition 4.3</u> [Σ-p-homomorphism]

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

$h = \{h_s: A_s \to B_s \mid s \in S\}$

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

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

Fact 4.4 [p-homomorphisms are closed under composition]

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

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

is a Σ-p-homomorphism.

Definition 4.5 [PEAlg]

The functor

PEAlg: SIG $\to$ 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).

Definition 4.6 [$Partial_\Sigma$]

For Σ ε SIG

$Partial_\Sigma$: EAlg(Σ) $\to$ PEAlg(Σ)

is the inclusion functor.

Fact 4.7 [Partial is a natural transformation]

Partial: EAlg ==> PEAlg

is a natural transformation.

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. Morphisms in that category are pairs of homomorphisms such that they commute with the abstraction functions. Similar to i-signatures, this situation can be expressed neatly as a comma category.

Definition 4.8 [Tripel(IΣ)]

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

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

is called the category of IΣ-tripels.

For an IΣ-tripel TA = <Ac,α,Aa>, Aa is called the abstract (or implemented) algebra, Ac the concrete (or implementing) algebra, and α is the abstraction function of TA.

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 4.9 [Tripel(τ)]

Let $\tau = \langle\rho a, \rho c\rangle: I\Sigma_1 \to I\Sigma_2 \in ISIG$.
$Tripel(\tau): Tripel(I\Sigma_2) \to Tripel(I\Sigma_1)$
defined on objects by
$$Tripel(\tau)(\langle Ac,\alpha,Aa\rangle) := \langle EAlg(\rho c)(Ac), PEAlg(\rho a)(\alpha), EAlg(\rho a)(Aa)\rangle$$
and on morphisms by
$$Tripel(\tau)(\langle hc,ha\rangle) := \langle EAlg(\rho c)(hc), EAlg(\rho a)(ha)\rangle$$
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 4.10 [IMod(IΣ)]

For every IΣ ε ISIG the category of IΣ-implementation models (or just IΣ-i-models)
$IMod(I\Sigma)$
is the full subcategory of Tripel(IΣ) generated by all tripels with surjective abstraction function.

Fact 4.11 [IMod(τ)]

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

Definition 4.12 [IMod]

IMod: ISIG → CAT^{op}
is the modelling functor for implementation signatures.

4.3 Relating implementation signatures to specifications

According to Section 2, implementation sentences over an i-signature IΣ 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.

With this decision we can define the set of IΣ-implementation sentences to be the set of all ordinary ψ(IΣ)-sentences, where ψ(IΣ) is a suitable equational signature combining Σa, Σc, and the abstraction operations.

We must define ψ(IΣ) such that the sentences which are to restrict the abstract algebras do not affect the concrete ones and vice versa. Therefore, taking the set theoretic union of signatures is not suitable, since the abstract and the concrete signatures need not be disjoint. Thus we take the disjoint union (or coproduct). Moreover, for reasons of convenience we will use standard names for the abstraction operations:

Definition 4.13 [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:
$$abs\text{-}operations(I\Sigma) := \{abs\text{-}s_{I\Sigma}: \sigma(s) \to s \mid s \in \Sigma a\}$$
$$abs\text{-}operations(\tau) := \{(abs\text{-}s_{I\Sigma}, abs\text{-}\rho a(s)_{I\Sigma'}) \mid s \in \Sigma a\}.$$

Fact 4.14 [ψ]

ψ: ISIG $\to$ SIG
defined on objects by
$\psi(I\Sigma) := \Sigma a \uplus \Sigma c \uplus$ abs-operations($I\Sigma$)
and on morphisms by
$\psi(\tau) := \rho a \uplus \rho c \uplus$ abs-operations(τ)
is a colimit preserving functor.

Having defined an $I\Sigma$-implementation sentence to be an ordinary $\psi(I\Sigma)$-sentence p we must determine whether an $I\Sigma$-i-model MA = ⟨Ac,α,Aa⟩ 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 4.15 [join $I\Sigma$(MA)]

For an i-signature $I\Sigma$ = ⟨SPa,σ,SPc⟩ and an $I\Sigma$-i-model MA = ⟨Ac,α,Aa⟩
$join_{I\Sigma}(MA) := Aa \uplus Ac \uplus \alpha$
is the $\psi(I\Sigma)$-algebra A defined by
- for s ε Sig(SPa): $A_s := Aa_s$
- for s ε Sig(SPc): $A_s := Ac_s$
- for op ε Sig(SPa): opA := opAa
- for op ε Sig(SPc): opA := opAc
- for abs-s ε abs-operations($I\Sigma$): 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 4.16 [join$I\Sigma$]

Defining $join_{I\Sigma}$ on $I\Sigma$-i-model morphisms g = ⟨hc,ha⟩ by
$join_{I\Sigma}(g) := \{ha_s \mid s \in Sig(SPa)\} \uplus \{hc_s \mid s \in Sig(SPc)\}$
yields a functor
$join_{I\Sigma}$: IMod($I\Sigma$) $\to$ EAlg($\psi(I\Sigma)$).

Generalizing over all i-signatures we obtain a natural transformation from the implementation model functor to the model functor of the SPEC-institution composed with the signature translation.

Fact 4.17

join: IMod ==⟩ EAlg $\circ$ ψ
is a natural transformation.

Now the question whether an $I\Sigma$-i-model MA satisfies an implementation sentence p has been reduced to the question whether $join_{I\Sigma}(MA)$ satisfies p in the framework of the SPEC-institution.

4.4 Implementation sentences and their satisfaction

According to the preceding section we define the set of $I\Sigma$-implementation sentences or just $I\Sigma$-i-sentences to be the set of all ordinary $\psi(I\Sigma)$-sentences. Such an $I\Sigma$-i-sentence p is satisfied by an $I\Sigma$-i-model MA exactly if MA viewed as the $\psi(I\Sigma)$-algebra $join_{I\Sigma}(MA)$ satisfies p.

Definition 4.18 [ISen]

The implementation sentence functor is given by
$ISen := Sen \circ \psi: ISIG \rightarrow SET.$

Definition 4.19 [$|\overset{i}{=}$]

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

Fact 4.20 [satisfaction condition]

$\forall\ \tau: I\Sigma_1 \rightarrow I\Sigma_2 \in ISIG\ .$
$\quad \forall\ MA \in IMod(I\Sigma_2)\ .$
$\quad\quad \forall\ p \in ISen(I\Sigma_1)\ .$
$$MA \overset{i}{\models}_{I\Sigma 2} ISen(\tau)(p) \iff IMod(\tau)(MA) \overset{i}{\models}_{I\Sigma 1} p.$$

4.5 The institution

Since the satisfaction condition holds the notions defined above constitute an institution.

Definition 4.21 [IMP-institution]

IMP-institution := $\langle ISIG, ISen, IMod, \overset{i}{\models} \rangle$
is the institution of implementation specifications.

Like specifications are defined as the theories of the SPEC-institution, implementation specifications will be defined as the theories of this new institution.

Definition 4.22 [IMP]

IMP is the category of theories of the IMP-institution and it is called the category of implementation specifications.

Thus an implementation specification or just i-specification ISP is a pair
$ISP = \langle I\Sigma, IE \rangle$
consisting of an i-signature $I\Sigma$ and a set of $I\Sigma$-i-sentences IE, and an i-specification morphism is an i-signature morphism respecting the i-sentences.

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

Fact 4.23 [colimits]

IMP is cocomplete.

5. Examples: developing imlementations of sets by lists

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 85]).

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.

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.

```
spec BOOL =
    sorts bool
    ops   true, false: → bool
          not: bool → bool
          and, or: bool bool → bool
    sentences ...  < specifying the booleans >

spec LIN-ORD = BOOL u
    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
```

Figure 5.1 The ADT specifications in the implementation of sets by lists

Presentations of the specifications mentioned so far are given in Figure 5.1. 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:

```
    ispec I:SET/LIST-S =
          isig σS/LS: SET → LIST-S
```

with the signature morphism

```
    σS/LS: Sig(SET) → SIG(LIST-S)
           set          → list
           empty        → nil
           empty?       → nil?
           insert       → l-insert
           in?          → l-in?
           min          → l-min
           remove-min   → l-remove-min
           x            → x        for x ε Sig(LIN-ORD)
```

It merely defines the signature morphism σS/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 implementations 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.

```
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 5.2 Some i-specifications implementing sets by lists

- all lists represent sets (IA:SET/LIST-S),
- only lists with unique entries may represent sets (IU:SET/LIST-S),
- only sorted lists may represent sets (IS:SET/LIST-S), or
- only sorted lists with unique 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 5.2 where we use abs-s: $\sigma_{S/LS}(s) \to s$ as the abstraction operation name of sort s.

By restricting the abstraction operations these alternative i-specifications constrain their i-models not only w.r.t. the abstraction function of sort set, but also w.r.t. the set simulating list operations. Correspondingly, we can specify four refinements of the concrete LIST-S specification by adding sentences fixing the set simulating operations, such that they generate (and operate upon) exactly
- all lists (LIST-SA),
- all lists with unique entries (LIST-SU),
- all sorted lists (LIST-SS), or
- all sorted lists with unique entries (LIST-SSU).

To give an example we elaborate the specification LIST-SS:

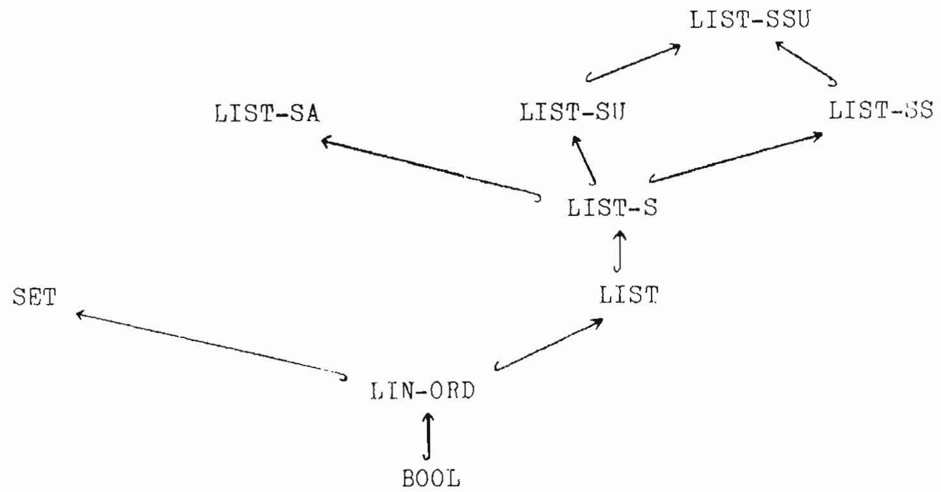
```
spec LIST-SS = LIST-S u
     sentences
          l-min(nil) = error-elem
          l-remove-min(nil) = nil
          (∀ e: elem . l-insert(e,nil) = cons(e,nil))
          (∀ x: list . ∀ e: elem .
               l-min(cons(e,x)) = e                                  &
               l-remove-min(cons(e,cons(e,x))) = l-remove-min(cons(e,x)) )
          (∀ x: list . ∀ e1, e2: elem .
               le(e1,e2) = true =>
                    l-insert(e1,cons(e2,x)) = cons(e1,cons(e2,x)) &
                    l-in?(e1,cons(e2,x)) = eq(e1,e2))             &
                    (eq(e1,e2) = false =>
                         l-insert(e2,cons(e1,x)) =  cons(e1,l-insert(e2,x)) &
                         l-in?(e2,cons(e1,x)) = l-in?(e2,x)                 &
                         l-remove-min(cons(e1,cons(e2,x))) = cons(e2,x)     ) )
```

Now we can in turn refine each of the i-specifications IX:SET/LIST-S for X ∈ {A, U, S, SU} by replacing the concrete specification LIST-S by its refinement LIST-XS and calling the resulting i-specifications IX:SET/LIST-SX. Since the abstraction operation of sort elem is not restricted, IX:SET/LIST-SX i-models have many non-isomorphic LIN-ORD implementations, e.g. the characters represented by the natural numbers or by the integers. However, for each LIN-ORD implementation there are only isomorphic IX:SET/LIST-SX i-models as extensions since the set simulating operations are fixed by now. Thus
- IA:SET/LIST-SA specifies the implementation of sets by all lists,
- IU:SET/LIST-SU specifies the implementation of sets by all lists with unique entries,
- IS:SET/LIST-SS specifies the implementation of sets by all sorted lists, and
- ISU:SET/LIST-SSU specifies the implementation of sets by all sorted lists with unique entries,

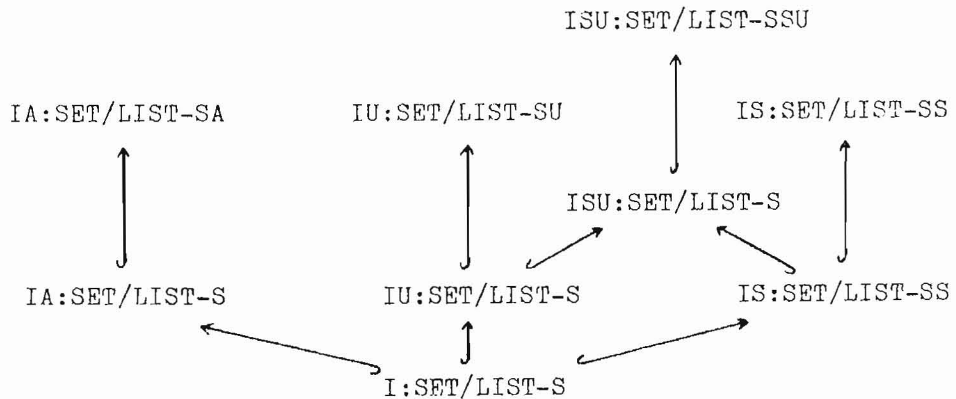
which are well known sets-by-lists implementations, differing in the time efficiency of the set simulating operations and in the amount of storage needed by the lists. As an example we give the i-specification IS:SET/LIST-SS:

<u>ispec</u> IS:SET/LIST-SS = IS:SET/LIST-S u
    <u>isig</u> σS/LS: SET → LIST-SS

The refinement relations between the specifications and i-specifications described above are depicted in Figures 5.3 and 5.4:



<u>Figure 5.3</u> The relations between the specifications



<u>Figure 5.4</u> The relations between the i-specifications

6. Conclusions

We proposed an implementation concept for loose abstract data type specifications. It introduces the notions of implementation signatures, - models, and - specifications and it formalizes the transition from a more abstract to a more concrete specification including a change of the underlying data structures. By providing the notion of refinement between implementations it supports to develop specifications and implementations hand in hand. Moreover, using institutions our concept abstracts from the underlying ADT specification method and is applicable to different specification techniques.

Other implementation concepts for loose specifications lack the notion of refinement. 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. 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 [BV 85] the implementation concept proposed here is elaborated for a particular institution of ADT specifications. An effective procedure is given to convert an implementation specification to a normal form which essentially consists of an ordinary ADT specification. Referring to the normal forms associative vertical composition operations for implementations are defined compatibly on the syntactical and semantical levels. 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

[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.

[BV 85] Beierle, C., Voß, A.: Algebraic specifications and implementations in an integrated software development and verfication system. FB Informatik, Univ. Kaiserslautern (to appear 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.

[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.

[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.

[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.

[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.