INFORMATION AND COMPUTATION 76, 165–210 (1988)

Specifications in an Arbitrary Institution*

DONALD SANNELLA AND ANDRZEJ TARLECKI†

Department of Computer Science, University of Edinburgh, Edinburgh, EH9 3JZ Scotland

A formalism for constructing and using axiomatic specifications in an arbitrary logical system is presented. This builds on the framework provided by Goguen and Burstall's work on the notion of an *institution* as a formalisation of the concept of a logical system for writing specifications. We show how to introduce free variables into the sentences of an arbitrary institution and how to add quantifiers which bind them. We use this foundation to define a set of primitive operations for building specifications in an arbitrary institution based loosely on those in the ASL kernel specification language. We examine the set of operations which results when the definitions are instantiated in institutions of total and partial first-order logic and compare these with the operations found in existing specification languages. We present proof rules which allow proofs to be conducted in specifications built using the operations we define. Finally, we introduce a simple mechanism for defining and applying parameterised specifications and briefly discuss the program development process.

1. INTRODUCTION

Much work has been done on algebraic specifications in the past 10 years. Although much has been accomplished, there is still no general agreement on the definitions of many of the basic concepts, e.g., signature and algebra, and on which kinds of axioms should be used. The disagreement arises partly because different definitions are required to treat various special issues in specification, such as errors [Gog 77, GDLE 82], coercions [Gog 78] and partial operations [BrW 82]; partly because some specification methods such as the initial algebra approach [GTW 76] only work under certain restrictions on, e.g., the form of axioms in specifications; and partly because of disagreements over matters of style or taste. These fundamental differences lead to difficulty in comparing the results achieved by different approaches and in building upon the work of others.

The notion of an *institution* [GB 84a] provides a tool for unifying all

* This is an essentially revised and extended version of (Sannella and Tarlecki, 1984).

† Present address: Institute of Computer Science, Polish Academy of Sciences, Warsaw.

0890-5401/88 $3.00

these different approaches to specification by formalising the concept of a logical system for writing specifications. An institution comprises definitions of signature, model, sentence (i.e., axiom), and satisfaction which obey a few internal consistency conditions (details in Section 2). Although it is often not obvious, much of the work which has been done on algebraic specification turns out to be independent of the particular definitions of these four notions. In such cases it would be highly desirable to make the generality explicit by basing everything on an arbitrary institution. This was done in the semantics of the Clear specification language [BG 80] (where an institution was called a "language"). Sometimes additional assumptions about the base institution are necessary, as in Clear where use of the initial algebra approach requires the assumption that the institution is liberal (forgetful functors induced by theory morphisms have left adjoints).[1] Instantiating the base institution in different ways (and changing the low-level syntax accordingly) yields a family of specification languages: equational Clear, error Clear, continuous Clear, and so on.

In early work on algebraic specification (e.g., [GTW 76]) it was shown how a collection of algebras could be specified by a *theory*, i.e., a signature together with a set of axioms. For small specifications such an approach is adequate, but it is more convenient to build large and complex specifications in a structured way by putting together small specifications. Several specification languages in addition to Clear support such a structured approach to specification. These include CIP-L [Bau 81], Look [ZLT 82, ETLZ 82], ASL [Wir 82, SW 83, Wir 83], the constraint language of [EWT 83], and the Larch Shared Language [GH 83]. None of these other languages were based on an arbitrary institution (although the possibility of a similar such generalisation was considered in [SW 83] and [EWT 83]) and so they are not general in the sense that Clear is. However, since they include features which seem desirable but which are not included in Clear, they are more useful as tools for writing specifications in the particular institutions they treat. Most useful of all would be an institution-based specification approach which incorporates the good ideas of all these languages. That is the goal of this paper. We define and carefully analyse a set of general specification-building operations based loosely (but not exclusively) on those in ASL.

One novel feature of ASL is a specification-building operation **abstract** which can be used to *behaviourally abstract* from a specification, closing its collection of models under behavioural equivalence [GGM 76, Rei 81]. This allows *abstract model specifications* [LB 77], cf. [Suf 82], in which the desired behaviour is described in some concrete way, e.g., by giving a

[1] On this point, see the "technical digression" in Section 5.

simple model which exhibits it. Such an operation is a necessary ingredient in an algebraic specification language since the specification of, e.g., an abstract data type is supposed to describe a behaviour (an input/output relation) without regard to the particular representation used and therefore *all* algebras which realise the desired behaviour should be permitted as models. However, this operation was defined in [SW 83] in such a way that it is not obvious how to generalise it to an arbitrary institution. (There are some remarks in [SW 83] which suggest how this might be done, but the proposed generalisation does not fit smoothly into the institutional framework and anyway the technical details are wrong.)

The key to the institution-based definition of **abstract** turns out to be the introduction of free variables into the sentences of the institution. We show in Section 3 how this may be accomplished. Free variables are necessary because they provide a way of naming unreachable elements of models which cannot be referred to using the operations of the model alone. Such elements play an important role in the definition of behavioural abstraction. Having introduced free variables into the sentences of an institution, we digress in the second part of Section 3 and show how to add quantifiers which bind them. This gives a construction for introducing quantified variables into the sentences of an arbitrary institution.

Building on this foundation, we then define a set of primitive operations for building specifications in an arbitrary institution (Section 4). The set of operations we provide is based on those present in ASL; however, there are a number of significant differences. These derive both from difficulties in generalising some of the operations of ASL to an arbitrary institution (for example, since we cannot easily form the union of signatures in this setting the + operation is not generalised directly) and from extensions which arose naturally in the process of generalisation. A feature of ASL which remains is the expressive power and flexibility necessary to provide a kernel for building high-level specification languages. The convenient-to-use specification-building operations of the high-level language would be defined by composing these low-level operations (as for example in PLUSS [Gau 84] built over ASL, and in Extended ML [ST 85] built over the specification-building operations presented here). It is natural for such high-level languages to hide some of the raw power of the primitives from the user.

It is worth noting that specifications may themselves be viewed as logical sentences, written in a (more expressive) logical system developed over the underlying institution. In fact, it is easy to see that specifications built using our specification-building operations form an institution.

In Section 5 we examine the set of operations which results when the general definitions are instantiated in an institution of first-order logic with equality as the only predicate and in an institution of partial first-order

logic. These operations are compared with those found in existing specification languages.

In Section 6 we consider the problem of theorem proving in the context of specifications built using the operations defined in Section 4. Following [SB 83], we present an approach which enables the structure of proofs to reflect and exploit the structure of specifications. For each of the specification-building operations we provide inference rules which are independent of the particular institution in use, show that they are sound, and analyse their completeness. This is another case were, due to the quest for generality via institutions, something (part of a theorem prover) may be built once and for all.

In Section 7 we introduce a mechanism for defining and applying parameterised specifications. In contrast to the usual way in which parameterised specifications are dealt with based on a pushout construction (see, e.g., [BG 80] and [Ehr 79]), we adopt a different approach based on the mechanism of macro-expansion (β-conversion in the λ-calculus). Finally, Section 8 concludes with remarks concerning the development of programs from specifications by stepwise refinement in this framework and the generality of our approach.

We assume some familiarity with a few notions from basic category theory, although no use is made of any deep results. See [AM 75, MacL 71] for the definitions which we omit here.

2. Institutions

Any approach to algebraic specification must be based on some logical framework. The pioneering papers [GTW 76, Gut 75, Zil 74] used many-sorted equational logic for this purpose. Nowadays, however, examples of logical systems in use include first-order logic (with and without equality), Horn-clause logic, higher-order logic, infinitary logic, temporal logic, and many others. Note that all these logical systems may be considered with or without predicates, admitting partial operations or not. This leads to different concepts of signature and of model, perhaps even more obvious in examples like polymorphic signatures, order-sorted signatures, continuous algebras, or error algebras.

There is no reason to view any of these logical systems as superior to the others; the choice must depend on the particular area of application and may also depend on personal taste. Another reason for choosing a particular logical system to work in might be because useful tools are available which only work in that framework (e.g., the availability of a Knuth–Bendix theorem prover might be an argument for working in equational logic).

The informal notion of a logical system has been formalised by Goguen

and Burstall [GB 84a], who introduced for this purpose the notion of an *institution*. An institution consists of a collection of signatures together with for any signature Σ a set of Σ-sentences, a collection of Σ-models, and a satisfaction relation between Σ-models and Σ-sentences. Note that signatures are arbitrary abstract objects in this approach, not necessarily the usual "algebraic" signatures used in many standard approaches to algebraic specification (see, e.g., [GTW 76]). The only "semantic" requirement is that when we change signatures, the induced translations of sentences and models preserve the satisfaction relation. This condition expresses the intended independence of the meaning of a specification from the actual notation. Formally:

DEFINITION [GB 84a]. An *institution* INS consists of

— a category $\mathbf{Sign}_{\mathrm{INS}}$ (of signatures),

— a functor $\mathbf{Sen}_{\mathrm{INS}}: \mathbf{Sign}_{\mathrm{INS}} \to \mathbf{Set}$ (where **Set** is the category of all sets; $\mathbf{Sen}_{\mathrm{INS}}$ gives for any signature Σ the set of Σ-sentences and for any signature morphism $\sigma: \Sigma \to \Sigma'$ the function $\mathbf{Sen}_{\mathrm{INS}}(\sigma): \mathbf{Sen}_{\mathrm{INS}}(\Sigma) \to \mathbf{Sen}_{\mathrm{INS}}(\Sigma')$ translating Σ-sentences to Σ'-sentences),

— a functor $\mathbf{Mod}_{\mathrm{INS}}: \mathbf{Sign}_{\mathrm{INS}} \to \mathbf{Cat}^{\mathrm{op}}$ (where **Cat** is the category of all categories;[2] $\mathbf{Mod}_{\mathrm{INS}}$ gives for any signature Σ the category of Σ-models and for any signature morphism $\sigma: \Sigma \to \Sigma'$ the σ-reduct functor $\mathbf{Mod}_{\mathrm{INS}}(\sigma)$: $\mathbf{Mod}_{\mathrm{INS}}(\Sigma') \to \mathbf{Mod}_{\mathrm{INS}}(\Sigma)$ translating Σ'-models to Σ-models), and

— a satisfaction relation $\models_{\Sigma,\mathrm{INS}} \subseteq |\mathbf{Mod}_{\mathrm{INS}}(\Sigma)| \times \mathbf{Sen}_{\mathrm{INS}}(\Sigma)$ for each signature Σ

such that for any signature morphism $\sigma: \Sigma \to \Sigma'$ the translations $\mathbf{Mod}_{\mathrm{INS}}(\sigma)$ of models and $\mathbf{Sen}_{\mathrm{INS}}(\sigma)$ of sentences preserve the satisfaction relation; i.e., for any $\varphi \in \mathbf{Sen}_{\mathrm{INS}}(\Sigma)$ and $M' \in |\mathbf{Mod}_{\mathrm{INS}}(\Sigma')|$

$$M' \models_{\Sigma',\mathrm{INS}} \mathbf{Sen}_{\mathrm{INS}}(\sigma)(\varphi) \quad \text{iff} \quad \mathbf{Mod}_{\mathrm{INS}}(\sigma)(M') \models_{\Sigma,\mathrm{INS}} \varphi$$

(Satisfaction condition).

To be useful as the underlying institution of a specification language, an institution must provide some tools for "putting things together." Thus, in this paper we additionally require that the category **Sign** has pushouts and initial objects (i.e., is finitely cocomplete) and moreover that **Mod** preserves pushouts and initial objects (and hence finite colimits), i.e., that **Mod** trans-

[2] Of course, some foundational difficulties are connected with the use of this category, as discussed in [MacL 71]. We do not discuss this point here, and we disregard other such foundational issues in this paper; in particular, we sometimes use the term "collection" to denote "sets" which may be too large to really be sets.

lates pushouts and initial objects in **Sign** to pullbacks and terminal objects (respectively) in **Cat**.

In [GB 84a] the category **Sign** is not required to be cocomplete, but this is required there of any institution to be used as the basis of a specification language (as in Clear [BG 80]). **Mod** is not required there to preserve colimits; however, we feel that this is a natural assumption to make the semantics of specification-building operations consistent with our intuitions. A similar condition is required in [EWT 83]. Note that both of these requirements are entirely independent of the "logical" part of the institution, i.e., of sentences and the satisfaction relation, and the fact that all standard examples of institutions (including all those in [GB 84a]) satisfy them indicates that they are not very restrictive in practice.

The work of [Bar 74] on abstract model theory is similar in intent to the theory of institutions but the notions used and the conditions they must satisfy are more restrictive and rule out many of the examples we would like to deal with.

Notational Conventions

— The subscript INS is omitted when there is no danger of confusion.

— We will write $\models$ instead of $\models_{\Sigma}$ when Σ is obvious.

— For any signature morphism $\sigma\colon \Sigma \to \Sigma'$, $\mathbf{Sen}(\sigma)$ is denoted just by σ and $\mathbf{Mod}(\sigma)$ is denoted by $-|_{\sigma}$ (i.e., for $\varphi \in \mathbf{Sen}(\Sigma)$, $\sigma(\varphi)$ stands for $\mathbf{Sen}(\sigma)(\varphi)$, and for $M' \in |\mathbf{Mod}(\Sigma')|$, $M'|_{\sigma}$ stands for $\mathbf{Mod}(\sigma)(M')$).

— For any signature Σ, $\Phi \subseteq \mathbf{Sen}(\Sigma)$ and $M \in |\mathbf{Mod}(\Sigma)|$, we write $M \models \Phi$ to denote that $M \models \varphi$ for all $\varphi \in \Phi$.

All of the examples of logical systems mentioned at the beginning of this section (e.g., first-order logic, temporal logic) fit into the framework of an institution. The following simple example serves to illustrate this. Note that we can define institutions which diverge from logical tradition and have, for example, sentences expressing constraints on models which are not usually considered in logic, e.g., data constraints as in Clear [BG 80], which may be used to impose the requirement of initiality (cf. [Rei 87, EWT 83]).

Example: The institution GEQ of ground equations

An *algebraic signature* is a pair $\langle S, \Omega \rangle$, where S is a set (of sort names) and Ω is a family of sets $\{\Omega_{w,s}\}_{w \in S^*, s \in S}$ (of operation names). We write $f\colon w \to s$ to denote $w \in S^*$, $s \in S$, $f \in \Omega_{w,s}$. An *algebraic signature morphism* σ:

$\langle S, \Omega \rangle \to \langle S', \Omega' \rangle$ is a pair $\langle \sigma_{\text{sorts}}, \sigma_{\text{opns}} \rangle$ where $\sigma_{\text{sorts}}: S \to S'$ and σ_{opns} is a family of maps $\{\sigma_{w,s}: \Omega_{w,s} \to \Omega'_{\sigma^*(w),\sigma(s)}\}_{w \in S^*, s \in S}$, where $\sigma^*(s1, ..., sn)$ denotes $\sigma_{\text{sorts}}(s1), ..., \sigma_{\text{sorts}}(sn)$ for $s1, ..., sn \in S$. We will write $\sigma(s)$ for $\sigma_{\text{sorts}}(s)$, $\sigma(w)$ for $\sigma^*(w)$, and $\sigma(f)$ for $\sigma_{w,s}(f)$, where $f \in \Omega_{w,s}$.

The category of algebraic signatures **AlgSig** has algebraic signatures as objects and algebraic signature morphisms as morphisms; the composition of morphisms is the composition of their corresponding components as functions. (This obviously forms a category.)

Let $\Sigma = \langle S, \Omega \rangle$ be an algebraic signature.

A *Σ-algebra* A consists of an S-indexed family of carrier sets $|A| = \{|A|_s\}_{s \in S}$ and for each $f: s1, ..., sn \to s$ a function $f_A: |A|_{s1} \times \cdots \times |A|_{sn} \to |A|_s$. A *$\Sigma$-homomorphism* from a Σ-algebra A to a Σ-algebra B, $h: A \to B$, is a family of functions $\{h_s\}_{s \in S}$, where $h_s: |A|_s \to |B|_s$, such that for any $f: s1, ..., sn \to s$ and $a_1 \in |A|_{s1}, ..., a_n \in |A|_{sn}$

$$h_s(f_A(a_1, ..., a_n)) = f_B(h_{s1}(a_1), ..., h_{sn}(a_n)).$$

The category of Σ-algebras $\mathbf{Alg}(\Sigma)$ has Σ-algebras as objects and Σ-homomorphisms as morphisms; the composition of homomorphisms is the composition of their corresponding components as functions. (This obviously forms a category.)

For any algebraic signature morphism $\sigma: \Sigma \to \Sigma'$ and Σ'-algebra A', the *σ-reduct* of A' is the Σ-algebra $A'|_\sigma$ defined as follows:

$$\text{For } s \in S,\ |A'|_\sigma|_s =_{\text{def}} |A'|_{\sigma(s)}.$$

$$\text{For } f: w \to s \text{ in } \Sigma,\ f_{A'|_\sigma} =_{\text{def}} \sigma(f)_{A'}.$$

Similarly, for a Σ'-homomorphism $h': A' \to B'$, where A' and B' are Σ'-algebras, the σ-reduct of h' is the Σ-homomorphism $h'|_\sigma: A'|_\sigma \to B'|_\sigma$ defined by $(h'|_\sigma)_s =_{\text{def}} h'_{\sigma(s)}$ for $s \in S$.

The mappings $A' \mapsto A'|_\sigma$, $h' \mapsto h'|_\sigma$ form a functor from $\mathbf{Alg}(\Sigma')$ to $\mathbf{Alg}(\Sigma)$.

For any algebraic signature Σ, $\mathbf{Alg}(\Sigma)$ contains an initial object T_Σ which is (to within isomorphism) the algebra of ground Σ-terms; i.e., the carriers $|T_\Sigma|$ contain terms of the appropriate sorts which are constructed using the operation symbols of Σ (without variables) and the operations in T_Σ are defined in the natural way (see, e.g., [GTW 76]). A *ground Σ-equation* is a pair $\langle t, t' \rangle$ (usually written as $t = t'$) where t, t' are ground Σ-terms of the same sort; i.e., $t, t' \in |T_\Sigma|_s$ for some sort s of Σ.

By definition, for any Σ-algebra A there is a unique Σ-homomorphism $h: T_\Sigma \to A$. For any ground term $t \in |T_\Sigma|_s$ (for s in the sorts of Σ) we write t_A rather than $h_s(t)$ to denote the value of t in A. For any Σ-algebra A and ground Σ-equation $t = t'$ we say that $t = t'$ *holds* in A (or A *satisfies* $t = t'$) written $A \models t = t'$, if $t_A = t'_A$.

Let $\sigma\colon \Sigma \to \Sigma'$ be an algebraic signature morphism. The unique Σ-homomorphism $h\colon T_\Sigma \to T_{\Sigma'}|_\sigma$ determines a translation of Σ-terms to Σ'-terms. For a ground Σ-term t of sort s we write $\sigma(t)$ rather than $h_s(t)$. This in turn determines a translation (again denoted by σ) of ground Σ-equations to ground Σ'-equations: $\sigma(t = t') =_{\text{def}} \sigma(t) = \sigma(t')$.

All the above notions combine to form the institution of ground equations GEQ:

— $\mathbf{Sign}_{\text{GEQ}}$ is the category of algebraic signatures **AlgSig**.

— For an algebraic signature Σ, $\mathbf{Sen}_{\text{GEQ}}(\Sigma)$ is the set of all ground Σ-equations; for an algebraic signature morphism $\sigma\colon \Sigma \to \Sigma'$, $\mathbf{Sen}_{\text{GEQ}}(\sigma)$ maps any ground Σ-equation $t = t'$ to the ground Σ'-equation $\sigma(t) = \sigma(t')$.

— For an algebraic signature Σ, $\mathbf{Mod}_{\text{GEQ}}(\Sigma)$ is $\mathbf{Alg}(\Sigma)$; for an algebraic signature morphism $\sigma\colon \Sigma \to \Sigma'$, $\mathbf{Mod}_{\text{GEQ}}(\sigma)$ is the functor $-|_\sigma\colon \mathbf{Alg}(\Sigma') \to \mathbf{Alg}(\Sigma)$.

— For an algebraic signature Σ, $\models_{\Sigma,\text{GEQ}}$ is the satisfaction relation as defined above.

It is easy to check that GEQ is an institution (the satisfaction condition is a special case of the satisfaction lemma of [BG 80]). The category **AlgSig** is finitely cocomplete (see [GB 84b, Prop. 5]) and $\mathbf{Mod}_{\text{GEQ}}\colon \mathbf{AlgSig} \to \mathbf{Cat}^{\text{op}}$ translates finite colimits in **AlgSig** to finite limits in **Cat** (see [BW 85]).

3. Free Variables in Institutions

In logic, formulae may contain free variables (such formulae are called *open*). To interpret an open formula, we must provide not only an interpretation for the symbols of the underlying signature (a model) but also an interpretation for the free variables (a valuation of variables into the model). This provides a natural way to deal with quantifiers. The need for open formulae also arises in the study of specification languages. In fact, we will need them to define one of the specification-building operations (**abstract**) in the next section. But for this we need institutions in which sentences may contain free variables.

Fortunately we do not have to change the notion of an institution—we can provide open formulae in the present framework (this idea was influenced by the treatment of variables in [Bar 74]). Note that we use here the term "formula" rather than "sentence," which is reserved for the sentences of the underlying institution.

Consider the institution GEQ of ground equations. Let $\Sigma = \langle S, \Omega \rangle$ be an algebraic signature. For any S-indexed family of sets, $X = \{X_s\}_{s \in S}$, define $\Sigma(X)$ to be the extension of Σ by the elements of X as new constants of the appropriate sorts.

Now, any sentence over $\Sigma(X)$ may be viewed as an open formula over Σ with free variables X. Given a Σ-algebra A, to determine whether an open Σ-formula with variables X holds in A we must first fix a valuation of variables X into $|A|$. Such a valuation corresponds exactly to an expansion of A to a $\Sigma(X)$-algebra, which additionally contains an interpretation of the constants X.

Given a translation of sentences along an algebraic signature morphism $\sigma\colon \Sigma \to \Sigma'$ we can extend it to a translation of open formulae. Roughly, we translate an open Σ-formula with variables X, which is a $\Sigma(X)$-sentence, to the corresponding $\Sigma'(X')$-sentence, which is an open Σ'-formula with variables X'. Here X' results from X by an appropriate renaming of sorts determined by σ (we also must avoid unintended "clashes" of variables and operation symbols).

The above ideas generalise to an arbitrary institution INS.

Let Σ be a signature.

Any pair $\langle \varphi, \theta \rangle$, where $\theta\colon \Sigma \to \Sigma'$ is a signature morphism and $\varphi \in \mathbf{Sen}(\Sigma')$, is an open Σ-formula with variables "$\Sigma' - \theta(\Sigma)$". (Note the quotation marks—since $\Sigma' - \theta(\Sigma)$ makes no sense in an arbitrary institution, it is only meaningful as an aid to our intuition.) When we use open formulae in specifications we will omit θ if it is obvious from the context.

If M is a Σ-model, $M \in |\mathbf{Mod}(\Sigma)|$, then a valuation of variables "$\Sigma' - \theta(\Sigma)$" into M is a Σ'-model $M' \in |\mathbf{Mod}(\Sigma')|$ which is a θ-expansion of M, i.e., $M'|_\theta = M$.

Note that in the standard logical framework there may be no valuation of a set of variables into a model containing an empty carrier. Similarly, here a valuation need not always exist (although there may be more reasons for that). For example, in GEQ if θ is not injective then some models have no θ-expansion.

If $\sigma\colon \Sigma \to \Sigma 1$ is a signature morphism and $\langle \varphi, \theta \rangle$ is an open Σ-formula then we define the translation of $\langle \varphi, \theta \rangle$ along σ as $\sigma(\langle \varphi, \theta \rangle) =_{\text{def}} \langle \sigma'(\varphi), \theta' \rangle$, where

$$\begin{array}{ccc} \Sigma' & \xrightarrow{\sigma'} & \Sigma 1' \\ {\scriptstyle\theta}\uparrow & & \uparrow{\scriptstyle\theta'} \\ \Sigma & \xrightarrow[\sigma]{} & \Sigma 1 \end{array}$$

is a pushout in the category of signatures.

There is a rather subtle problem we must point out here: pushouts are defined only up to isomorphism, so strictly speaking the translation of open formulae is not well-defined. Fortunately, from the definition of an institution one may easily prove that whenever $\iota\colon \Sigma 1' \to \Sigma 1''$ is an

isomorphism in **Sign** with inverse ι^{-1} then $\mathbf{Sen}(\iota)$: $\mathbf{Sen}(\Sigma 1') \to \mathbf{Sen}(\Sigma 1'')$ is a bijection, $\mathbf{Mod}(\iota)$: $\mathbf{Mod}(\Sigma 1'') \to \mathbf{Mod}(\Sigma 1')$ is an isomorphism in **Cat**, and moreover for any $\Sigma 1'$-sentence $\psi \in \mathbf{Sen}(\Sigma 1')$ and any $\Sigma 1'$-model $M1' \in |\mathbf{Mod}(\Sigma 1')|$,

$$M1' \models \psi \quad \text{iff} \quad M1'|_{\iota^{-1}} \models \iota(\psi).$$

This shows that (at least for semantic analysis) we can pick out an arbitrary pushout to define the translation of open formulae and so we may safely accept the above definition of translation.

Note that sometimes we want to restrict the class of signature morphisms which may be used (as second components) to construct open formulae. In fact, in the above remark sketching how free variables may be introduced into GEQ we used only algebraic signature inclusions $\iota: \Sigma \to \Sigma'$, where the only new symbols in Σ' were constants. To guarantee that the translation of open formulae is defined under such a restriction, we consider only restrictions to a collection $\mathscr{M}$ of signature morphisms which is closed (at least) under pushing out along arbitrary signature morphisms, i.e., for any signature morphism $\sigma: \Sigma \to \Sigma 1$ if $\theta: \Sigma \to \Sigma' \in \mathscr{M}$ then there is a pushout in **Sign**

$$\begin{array}{ccc} \Sigma' & \xrightarrow{\sigma'} & \Sigma 1' \\ {\scriptstyle\theta}\uparrow & & \uparrow{\scriptstyle\theta'} \\ \Sigma & \xrightarrow[\sigma]{} & \Sigma 1 \end{array}$$

such that $\theta' \in \mathscr{M}$.

Examples of such collections $\mathscr{M}$ in **AlgSig** include the collection of all algebraic signature inclusions, the restriction of this to inclusions $\theta: \Sigma \to \Sigma'$ such that Σ' contains no new sorts, the further restriction of this by the requirement that Σ' contains new constants only (as above), the collection of all algebraic signature morphisms which are onto w.r.t. sorts, the collection of all identities, and the collection of all morphisms. Note that most of these permit variables denoting operations or even sorts.

In the rest of this section we briefly sketch how to existentially close the open formulae introduced above (the construction is based on the notion of a *syntactic operation* in [Bar 74]). It is therefore a bit peripheral to the main concern of this paper but we would like to add some logical meat to our treatment of free variables.

Let $\mathscr{M}$ be a collection of signature morphisms which is closed under pushing out along arbitrary morphisms in **Sign**. Let Σ be a signature and let $\langle \varphi, \theta \rangle$ be an open Σ-formula such that $\theta \in \mathscr{M}$. Consider the existential closure of $\langle \varphi, \theta \rangle$, written $\exists \langle \varphi, \theta \rangle$, as a new Σ-sentence. The satisfaction

relation and the translation of sentences $\exists\langle\varphi, \theta\rangle$ along a signature morphism are defined in the expected way:

— A Σ-model satisfies $\exists\langle\varphi, \theta\rangle$ if it has a θ-expansion which satisfies φ; i.e., for any $M \in |\mathbf{Mod}(\Sigma)|$,

$M \models \exists\langle\varphi, \theta\rangle$ iff there exists $M' \in |\mathbf{Mod}(\Sigma')|$ such that $M'|_\theta = M$ and $M' \models \varphi$.

— For any signature morphism σ: $\Sigma \to \Sigma 1$, $\sigma(\exists\langle\varphi, \theta\rangle) =_{\text{def}} \exists\sigma(\langle\varphi, \theta\rangle)$, where $\sigma(\langle\varphi, \theta\rangle) = \langle\sigma'(\varphi), \theta'\rangle$ is the translation of $\langle\varphi, \theta\rangle$ as an open Σ-formula (with $\theta' \in \mathscr{M}$).

Note that in the above we have extended our underlying institution INS. Formally, we can define the extension of INS by existential closure w.r.t. $\mathscr{M}$, $\mathrm{INS}^{\exists}(\mathscr{M})$, to be the following institution:

— $\mathbf{Sign}_{\mathrm{INS}^{\exists}(\mathscr{M})}$ is $\mathbf{Sign}_{\mathrm{INS}}$.

— For any signature Σ, $\mathbf{Sen}_{\mathrm{INS}^{\exists}(\mathscr{M})}(\Sigma)$ is the disjoint union of $\mathbf{Sen}_{\mathrm{INS}}(\Sigma)$ with the collection of all existential closures $\exists\langle\varphi, \theta\rangle$ of open Σ-formulae, where $\theta \in \mathscr{M}$; for a signature morphism $\sigma: \Sigma \to \Sigma 1$, $\mathbf{Sen}_{\mathrm{INS}^{\exists}(\mathscr{M})}(\sigma)$ is the function induced by $\mathbf{Sen}_{\mathrm{INS}}(\sigma)$ on $\mathbf{Sen}_{\mathrm{INS}}(\Sigma)$ and by the notion of translation of existentially closed open formulae as defined above.

— $\mathbf{Mod}_{\mathrm{INS}^{\exists}(\mathscr{M})}$ is $\mathbf{Mod}_{\mathrm{INS}}$.

— The satisfaction relation in $\mathrm{INS}^{\exists}(\mathscr{M})$ is induced by the satisfaction relation of INS for INS-sentences and the notion of satisfaction for existentially closed open formulae as defined above.

The following theorem guarantees that $\mathrm{INS}^{\exists}(\mathscr{M})$ is in fact an institution (modulo the above remark about translations of open formulae).

THEOREM (Satisfaction condition for $\mathrm{INS}^{\exists}(\mathscr{M})$). *For any signature morphism* σ: $\Sigma \to \Sigma 1$, *open* Σ*-formula* $\langle\varphi, \theta\rangle$ *and* $\Sigma 1$*-model* $M1 \in |\mathbf{Mod}(\Sigma 1)|$,

$$M1|_\sigma \models \exists\langle\varphi, \theta\rangle \quad \textit{iff} \quad M1 \models \sigma(\exists\langle\varphi, \theta\rangle).$$

Proof. Recall that $\sigma(\exists\langle\varphi, \theta\rangle) = \exists\langle\sigma'(\varphi), \theta'\rangle$, where

$$\begin{array}{ccc} \Sigma' & \xrightarrow{\sigma'} & \Sigma 1' \\ {\scriptstyle\theta}\uparrow & & \uparrow{\scriptstyle\theta'} \\ \Sigma & \xrightarrow[\sigma]{} & \Sigma 1 \end{array}$$

is a pushout in **Sign**.

In the proof we need the following lemma, which is a consequence of our

assumption that the functor **Mod** translates pushouts in **Sign** to pullbacks in **Cat** (we omit the obvious proof based on the construction of pullbacks in **Cat**).

LEMMA. *For any two models* $M1 \in |\mathbf{Mod}(\Sigma 1)|$ *and* $M' \in |\mathbf{Mod}(\Sigma')|$ *such that* $M1|_\sigma = M'|_\theta$ *there is a unique model* $M1' \in |\mathbf{Mod}(\Sigma 1')|$ *such that* $M1'|_{\sigma'} = M'$ *and* $M1'|_{\theta'} = M1$.

The proof of the theorem is now straightforward:

($\Rightarrow$): Assume that $M1|_\sigma \models \exists\langle\varphi, \theta\rangle$. Thus, by the definition of the satisfaction relation for $\exists\langle\varphi, \theta\rangle$, there exists $M' \in |\mathbf{Mod}(\Sigma')|$ such that $M'|_\theta = M1|_\sigma$ and $M' \models \varphi$. By the above lemma, there is a $\Sigma 1'$-model $M1'$ such that $M1'|_{\sigma'} = M'$ and $M1'|_{\theta'} = M1$. Now, by the satisfaction condition we have $M1' \models \sigma'(\varphi)$. Hence, $M1$ has a θ'-expansion which satisfies $\sigma'(\varphi)$, i.e., $M1 \models \exists\langle\sigma'(\varphi), \theta'\rangle$.

($\Leftarrow$): Assume that $M1 \models \exists\langle\sigma'(\varphi), \theta'\rangle$. Thus, there is $M1' \in |\mathbf{Mod}(\Sigma 1')|$ such that $M1'|_{\theta'} = M1$ and $M1' \models \sigma'(\varphi)$. By the satisfaction condition, $M1'|_{\sigma'} \models \varphi$. Hence, $M1|_\sigma \models \exists\langle\varphi, \theta\rangle$ since $M1|_\sigma = M1'|_{\sigma;\theta'} = M1'|_{\theta;\sigma'} = (M1'|_{\sigma'})|_\theta$. ∎

Obviously, other quantifiers (for all, for almost all, there exists a unique, there exist infinitely many, ...) may be introduced to institutions in the same manner as we have just introduced existential quantifiers. In particular, in [Tar 86] it is shown in detail how to build over an arbitrary institution INS the institution of *universally* closed sentences, $\mathrm{INS}^\forall(\mathscr{M})$ (for any class $\mathscr{M}$ of signature morphisms as in the above construction of $\mathrm{INS}^\exists(\mathscr{M})$). It is also worth mentioning that one may similarly introduce logical connectives (cf. [Bar 74]). By iterating these ideas we can, for example, derive the institution of first-order logic from the institution of ground atomic formulae.

EXAMPLE. Let $\mathscr{I}$ be the collection of morphisms $\iota: \Sigma \to \Sigma'$ in **AlgSig** such that ι is an algebraic signature inclusion and Σ' contains new constants only. The institution $\mathrm{GEQ}^\exists(\mathscr{I})$ (resp. $\mathrm{GEQ}^\forall(\mathscr{I})$) is the institution of existentially quantified equations (resp. universally quantified equations—cf. [GB 84a]). If we additionally allow Σ' to contain new operation names (not just constants) then quantification along morphisms in $\mathscr{I}$ leads to a version of second-order logic.

4. BUILDING SPECIFICATIONS

Institutions provide an adequately general framework for dealing with basic problems such as what specifications are and how they may be built.

In attacking these problems below we assume that we are working in the framework of a fixed but arbitrary institution INS.

There are various levels at which specifications may be dealt with. We can consider

— textual level, a sequence of characters on paper (or some other storage medium);

— presentation level, a signature and a set of sentences (axioms) over this signature (required to be finite or at least recursive or recursively enumerable);

— theory level, a signature and a set of axioms over this signature closed under logical consequence;

— model level, a signature and a class of models over this signature.

Each approach to specification needs the textual level for actually writing down specifications. The meaning of a specification text is determined by giving a mapping from the textual level to one of the other levels. For example, specifications are mapped in Clear to theories, in ASL to classes of models, and in ACT ONE [EFH 83] to both presentations (the "first-level" semantics) and to classes of models (the "second-level" semantics). There are natural mappings from presentations to theories and from theories to classes of models (a presentation maps to the smallest theory containing it, and a theory maps to the class of models satisfying its axioms); the second-level semantics of ACT ONE is actually redundant since it is just the composition of the first-level semantics with these natural mappings, as proved in [EFH 83]. However, not every class of models is the class of models of a theory, and not every theory has a (finite, recursive, or recursively enumerable) presentation. For example, Clear has no presentation-level semantics and neither ASL nor the set of specification building operations presented below has a presentation- or theory-level semantics.

However, one thing which is certain is that each specification is a textual object describing a signature and a class of models over that signature (called the *models of the specification*). And this is in the end all that really matters since the purpose of a specification is not to describe a presentation or a theory but rather to describe a class of models (actually, to describe a class of programs, but models are just what you get when you abstract away from the syntactic and algorithmic details of programs).

To formalise this, for any specification SP we define its signature $\mathrm{Sig}[\mathrm{SP}] \in |\mathbf{Sign}|$ and the collection of its models $\mathrm{Mod}[\mathrm{SP}] \subseteq |\mathbf{Mod}(\mathrm{Sig}[\mathrm{SP}])|$. If $\mathrm{Sig}[\mathrm{SP}] = \Sigma$ then we call SP a *Σ-specification.*

We will not put any restrictions on the class of models described by a specification. Thus, specifications may be *loose* (having non-isomorphic models), so as to avoid premature design decisions. In contrast to many

approaches (e.g., CIP-L [Bau 81]) we do not require models to be *reachable* (in the standard framework, an algebra is reachable if every element is the value of some ground term; for the generalisation to an arbitrary institution see [Tar 85]). We do not even assume that the class of models of a specification is closed under isomorphism. On the other hand, these restrictions are not ruled out, and in fact we provide mechanisms to allow such restrictions to be included in specifications when required.

In order to make big specifications easy to understand and use, we build them in a structured way from small bits using *specification-building operations*. The semantics of each of these operations is a function on classes of models. A specification language may be viewed as a set of such operations, together with some syntax. Some operations correspond to functions at the presentation or theory level, but in general this need not be so—in any case they are described by functions at the model level.

In choosing the set of operations there is a trade-off between the expressive power of the language and the ease of understanding and dealing with the operations. One way to circumvent this problem is to first develop a *kernel* language which consists of a minimal set of very powerful operations. Such a kernel language is difficult to use directly. We can build higher-level languages on top of the kernel, so that each higher-level construct corresponds to a kernel-language expression. This is analogous to the way that high-level programming languages are defined in terms of machine-level operations.

In the rest of this section we describe a set of simple operations for building specifications in an arbitrary institution. Our intention is to provide low-level operations which collectively give sufficient power and flexibility to constitute a kernel for building high-level specification languages in any institution. In fact, we have already defined a high-level specification language called Extended ML [ST 85] on top of the operations described below. Another experiment of this kind is the language PLUSS [Gau 84] built on top of ASL.

We do not claim that the notion of a kernel is mathematically well-defined in any sense. In referring to the operations defined below as forming a kernel we wish only to convey that we regard them as low-level operations which are not necessarily convenient to use, and which form an appropriate foundation for building higher-level specification languages in the manner just described. It would be interesting to look for some way of evaluating the "kernelness" of a set of specification-building operations. This would presumably be based on the number of different specification-building operations which can be expressed using those in the kernel. Some work in this direction appears in [Wir 83] in which it is shown that (in the standard algebraic framework) ASL can be used to define any computable transformation of classes of algebras.

We intentionally do not define a formal specification language here but only the specification-building operations behind such a language. The difference is mainly one of syntax; although we provide a suggestive notation for our operations, this is not a complete syntax yet because without fixing a particular institution the syntax of signatures, signature morphisms, and sentences cannot be fixed. We also do not care to define a notation for describing sets. This attitude admits a certain informality in the presentation below. However, we do take care to formally define the semantics of all our operations. In [Wir 83] a complete syntax for ASL in the standard algebraic framework is developed.

4.1. *Specification-Building Operations*

The operations we provide are the following:

— Form a basic specification given a signature Σ and set Φ of Σ-sentences. This specifies the collection of Σ-models that satisfy Φ.

— Form the union of a family of Σ-specifications $\{SP_i\}_{i \in I}$, specifying the collection of Σ-models satisfying SP_i for all $i \in I$.

— Translate a Σ-specification to another signature Σ' along a signature morphism $\sigma: \Sigma \to \Sigma'$. This together with union allows large specifications to be built from smaller and more or less independent specifications.

— Derive a Σ'-specification from a specification over a richer signature Σ using a signature morphism $\sigma: \Sigma' \to \Sigma$. This allows details of a constructive specification to be hidden while essentially preserving its collection of models.

— Given a Σ-specification restrict models to only those which are minimal extensions of their σ-reducts for a given $\sigma: \Sigma' \to \Sigma$. This imposes on the models of a specification an additional constraint which excludes models which are "larger" than necessary.

— Close the collection of models of a specification under isomorphism.

— Abstract away from certain details of a specification, admitting any models which are equivalent to a model of the specification w.r.t. some given set of properties (defined using sentences of the institution).

We defer discussion of the **abstract** operation to the next subsection. Here is a more formal description of the other operations (we discuss their instantiations in two typical institutions at a more intuitive level in Section 5).

Basic specifications. Let $\Sigma \in |\mathbf{Sign}|$ be a signature and $\Phi \subseteq \mathbf{Sen}(\Sigma)$ a set

of Σ-sentences. The pair $\langle \Sigma, \Phi \rangle$ is then a specification with semantics defined as

$$\mathrm{Sig}[\langle \Sigma, \Phi \rangle] = \Sigma$$
$$\mathrm{Mod}[\langle \Sigma, \Phi \rangle] = \{M \in |\mathbf{Mod}(\Sigma)| \mid M \models \Phi\}.$$

When the signature Σ is obvious from the context, we will sometimes write Φ instead of $\langle \Sigma, \Phi \rangle$.

The union of a family of specifications. If $\{\mathrm{SP}_i\}_{i \in I}$ is a family of Σ-specifications (so $\mathrm{Sig}[\mathrm{SP}_i] = \Sigma$ for all $i \in I$) then $\bigcup_{i \in I} \mathrm{SP}_i$ is a specification with the following semantics,

$$\mathrm{Sig}\left[\bigcup_{i \in I} \mathrm{SP}_i\right] = \Sigma$$
$$\mathrm{Mod}\left[\bigcup_{i \in I} \mathrm{SP}_i\right] = \bigcap_{i \in I} \mathrm{Mod}[\mathrm{SP}_i]$$

(where $\bigcap$ denotes set-theoretic intersection). Note that if each of the specifications SP_i for $i \in I$ is a basic specification $\langle \Sigma, \Phi_i \rangle$ then their union has the same collection of models as $\langle \Sigma, \bigcup_{i \in I} \Phi_i \rangle$ (this time $\bigcup$ denotes the usual set-theoretic union). As usual, when I is finite we may use the usual infix notation; e.g., we may write $\mathrm{SP}_1 \cup \cdots \cup \mathrm{SP}_n$ if $I = \{1, \ldots, n\}$.

Translating a specification along a signature morphism. If SP is a Σ-specification and $\sigma: \Sigma \to \Sigma'$ is a signature morphism then **translate** SP **by** σ is a specification with semantics defined as

$$\mathrm{Sig}[\textbf{translate SP by } \sigma] = \Sigma'$$
$$\mathrm{Mod}[\textbf{translate SP by } \sigma] = \{M' \in |\mathbf{Mod}(\Sigma')| \mid |M'|_\sigma \in \mathrm{Mod}[\mathrm{SP}]\}.$$

If SP is a basic specification $\langle \Sigma, \Phi \rangle$ then **translate** SP **by** σ has the same collection of models as $\langle \Sigma', \sigma(\Phi) \rangle$, where $\sigma(\Phi)$ is the image of Φ under σ (i.e., $\mathbf{Sen}(\sigma)$).

Note again that using union we can only "put together" specifications with the same signature. To combine specifications with different signatures we must form a "union signature" (renaming some of the signature symbols if necessary), translate the specifications into this "union signature" (using **translate** w.r.t. appropriate signature injections), and then form the union of the translated specifications. All this may be combined into one operation using an appropriate category of "signature inclusions" to form the "union signature" as a coproduct (R. Burstall, private communication; cf. also a remark in [GB 84a, Sec. 6.1]). However, we decided to keep two

simple, more elementary operations (which gives slightly more flexibility) rather than provide a single higher-level operation.

Deriving one specification from another. If $\sigma: \Sigma' \to \Sigma$ is a signature morphism and SP is a Σ-specification then **derive from** SP **by** σ is a specification with the semantics

$$\mathrm{Sig}[\textbf{derive from SP by } \sigma] = \Sigma'$$

$$\mathrm{Mod}[\textbf{derive from SP by } \sigma] = \{M|_\sigma \mid M \in \mathrm{Mod}[\mathrm{SP}]\}.$$

For $\Phi \subseteq \mathbf{Sen}(\Sigma)$, $\mathrm{Mod}[\textbf{derive from } \langle \Sigma, \Phi \rangle \textbf{ by } \sigma] \subseteq \mathrm{Mod}[\langle \Sigma', \sigma^{-1}(\Phi) \rangle]$, where $\sigma^{-1}(\Phi)$ is the coimage of Φ under σ (i.e., $\mathbf{Sen}(\sigma)$). Note however that this inclusion may be proper, since sometimes not all the properties of models of the derived specification are expressible using just Σ'-sentences. Although the semantics of our **derive** is different from the semantics of the **derive** operation in Clear [BG 80] (which produces the model class on the right-hand side of the above inclusion provided that Φ is closed under consequence) we have chosen to use the same name. The difference between the two will be explored further in the next section.

Restricting to the minimal models of a specification. To define restriction to the minimal models of a specification we need the following notion:

Let $\sigma: \Sigma' \to \Sigma$ be a signature morphism and $K \subseteq |\mathbf{Mod}(\Sigma)|$ be a collection of Σ-models. We say that a model M is *σ-minimal in K* if $M \in K$ and if M contains (to within isomorphism) no proper submodel from K with an isomorphic σ-reduct, which we formalise as follows: for every $M1 \in K$, any monomorphism $m: M1 \to M$ (in $\mathbf{Mod}(\Sigma)$) such that $m|_\sigma$ is an isomorphism from $M1|_\sigma$ to $M|_\sigma$ (in $\mathbf{Mod}(\Sigma')$) is in fact an isomorphism (in $\mathbf{Mod}(\Sigma)$).

Now, for any signature morphism $\sigma: \Sigma' \to \Sigma$ and Σ-specification SP, **minimal** SP **wrt** σ is a specification describing the models of SP which are minimal extensions of their σ-reducts:

$$\mathrm{Sig}[\textbf{minimal SP wrt } \sigma] = \Sigma$$

$$\mathrm{Mod}[\textbf{minimal SP wrt } \sigma] = \{M \mid M \text{ is } \sigma\text{-minimal in } \mathrm{Mod}[\mathrm{SP}]\}.$$

Closing the model class of a specification under isomorphism. If SP is a Σ-specification then **iso close** SP is a specification with semantics defined by

$$\mathrm{Sig}[\textbf{iso close SP}] = \Sigma$$

$$\mathrm{Mod}[\textbf{iso close SP}] = \{M \in |\mathbf{Mod}(\Sigma)| \mid M \text{ is isomorphic to some model } M1 \in \mathrm{Mod}[\mathrm{SP}]\}.$$

Observe that there is no guarantee in the definition of an institution that

the satisfaction relation is preserved under isomorphism of models. Thus, even the collection of models of a basic specification need not be closed under isomorphism. Also note (see Section 5) that the collection of models of **derive from** SP **by** σ need not be closed under isomorphism even if the collection of models of SP is.

4.2. *Observational Abstraction*

A concept which has (not) been extensively (enough) studied in the context of algebraic specifications is that of the behaviour of a program or model. Intuitively, the behaviour of a program is determined just by the answers which are obtained from computations the program may perform. Switching for a while to the usual algebraic framework, we may say informally that two Σ-algebras are *behaviourally equivalent* with respect to a set OBS of *observable sorts* if it is not possible to distinguish between them by evaluating Σ-terms which produce a result of observable sort. For example, suppose Σ contains the sorts *nat*, *bool*, and *bunch* and the operations *empty*: $\rightarrow$ *bunch*, *add*: *nat*, *bunch* $\rightarrow$ *bunch*, and $\in$: *nat*, *bunch* $\rightarrow$ *bool* (as well as the usual operations on *nat* and *bool*), and suppose A and B are Σ-algebras with

$$|A_{\text{bunch}}| = \text{the set of finite sets of natural numbers}$$

$$|B_{\text{bunch}}| = \text{the set of finite lists of natural numbers}$$

with the operations and the remaining carriers defined in the obvious way (but B does *not* contain operations like *cons*, *car*, and *cdr*). Then A and B are behaviourally equivalent with respect to $\{bool\}$ since every term of sort *bool* has the same value in both algebras (the interesting terms are of the form $m \in add(a_1, ..., add(a_n, empty)...)$). Note that A and B are not isomorphic. Different notions of behavioural equivalence have been studied in [GGM 76, BM 81, Rei 81, GM 82, Sch 82, Kam 83, GM 83, SW 83] and elsewhere; the idea goes back (at least) to work on automata theory in the 1950s [Moore 56].

Behavioural equivalence seems to be a concept which is fundamental to programming methodology. For example:

Data abstraction. A practical advantage of using abstract data types in the construction of programs is that the implementation of abstractions by program modules need not be fixed. A different module using different algorithms and/or different data structures may be substituted without changing the rest of the program provided that the new module is behaviourally equivalent to the module it replaces (with respect to the non-encapsulated types). ADJ [GTW 76] have suggested that "abstract" in

"abstract data type" means "up to isomorphism"; we suggest that it really means "up to behavioural equivalence."[3]

Program specification. One way of specifying a program is to describe the desired input/output behaviour in some concrete way, e.g., by constructing a very simple program which exhibits the desired behaviour. Any program which is behaviourally equivalent to the sample program with respect to the primitive types of the programming language satisfies the specification. This is called an *abstract model specification* [LB 77] or *specification by example* [Sad 84]. In general, specifications under the usual algebraic approaches are not abstract enough; it is either difficult, as in Clear [BG 80], or impossible, as in the initial algebra approach of [GTW 76] and the final algebra approach of [Wand 79] to specify sets of natural numbers in such a way that both A and B above are models of the specification. One way to do specification by example in our framework is to use a *behavioural abstraction* operation which when applied to a specification SP relaxes interpretation to all those algebras which are behaviourally equivalent to a model of SP. We want to stress that although the phrase "specification by example" suggests sloppiness, this is not the case; in this approach it is a precisely defined, convenient, and intuitive way to write specifications, and it is also an established technique in software engineering.

In the above we assume that the only observations (or experiments) we are allowed to perform are to test whether the results of computations are equal. In the context of an arbitrary institution we can generalise this and abstract away from the equational bias by allowing observations which are arbitrary sentences (logical formulae). This yields an institution-based notion of *observational equivalence.* Two models are observationally equivalent if they both give the same answers to any observation from a prespecified set. Based on this general notion of observational equivalence we can define an institution-based specification-building operation for observational abstraction (the behavioural abstraction operation mentioned above is actually only a special case of observational abstraction in the standard algebraic framework). One complication is that in order to

[3] It is not our intention to quibble over terminology here. We only wish to suggest that the use of the word "abstract" in "abstract data type," meaning "independent of representation" according to [GTW 76], is more accurately reflected by the notion of behavioural equivalence than by isomorphism as was suggested there. This seems to be consistent with the use of the term in languages like CLU [Lis 81] (where abstract data types are called *clusters*). In [GM 82, 83] it has been suggested that "abstract data type" is an appropriate term for an isomorphism class of algebras while "abstract machine" refers to a behavioural equivalence class of algebras. Then a CLU cluster would correspond to an abstract machine. Since the motivation is really to capture algebraically the idea embodied in CLU clusters, we are in agreement with Goguen and Meseguer although we choose to use a different terminology.

deal with non-reachable models we must be able to express the observations we want to make as open formulae; the free variables provide a way of naming unreachable elements.

Formally, for any signature Σ, signature morphism $\theta: \Sigma \to \Sigma'$, set $\Phi' \subseteq \mathrm{Sen}(\Sigma')$ of open Σ-formulae with variables "$\Sigma' - \theta(\Sigma)$" and Σ-models $M1, M2 \in |\mathrm{Mod}(\Sigma)|$, we say that $M1$ is *observationally reducible to* $M2$ *w.r.t.* Φ' *via* θ, written $M1 \leqslant^{\theta}_{\Phi'} M2$, if for every (valuation) $M1' \in |\mathrm{Mod}(\Sigma')|$ with $M1'|_\theta = M1$ there exists (a valuation) $M2' \in |\mathrm{Mod}(\Sigma')|$ with $M2'|_\theta = M2$, such that for all $\varphi \in \Phi'$, $M1' \models \varphi$ iff $M2' \models \varphi$. $M1$ is *observationally equivalent to* $M2$ *w.r.t.* Φ' *via* θ, written $M1 \equiv^{\theta}_{\Phi'} M2$, if $M1 \leqslant^{\theta}_{\Phi'} M2$ and $M2 \leqslant^{\theta}_{\Phi'} M1$.

Now, for any Σ-specification SP, signature morphism $\theta: \Sigma \to \Sigma'$, and set $\Phi' \subseteq \mathbf{Sen}(\Sigma')$ of open Σ-formulae with variables "$\Sigma' - \theta(\Sigma)$," the specification **abstract** SP **wrt** Φ' **via** θ (intuitively) ignores the properties specified in SP as much as possible without affecting Φ'; i.e., it admits any model Φ'-equivalent via θ to a model of SP:

$$\mathrm{Sig}[\textbf{abstract SP wrt } \Phi' \textbf{ via } \theta] = \Sigma$$

$$\mathrm{Mod}[\textbf{abstract SP wrt } \Phi' \textbf{ via } \theta] = \{M1 \in |\mathbf{Mod}(\Sigma)| \mid M1 \equiv^{\theta}_{\Phi'} M2 \text{ for some } M2 \in \mathrm{Mod}[\mathrm{SP}]\}.$$

In Section 5 we describe how observational abstraction may be applied to give the effect of the behavioural abstraction operation mentioned above.

It is worth noting that specifications in a given institution INS themselves form an institution. This institution has the same signatures and models as INS. Sentences over a signature Σ are Σ-specifications as defined in Sections 4.1 and 4.2, and the translation of a Σ-"sentence" along a signature morphism $\sigma: \Sigma \to \Sigma'$ is defined in the obvious way using the **translate** operation. Note, however, that in order for this to form a functor we must "normalize" specifications so that they have at most one **translate** operation outside, i.e., we must identify **translate** (**translate** SP **by** σ) **by** σ' with **translate** SP **by** $\sigma; \sigma'$. The satisfaction relation is determined by the semantics of specifications: a Σ-model M satisfies a Σ-"sentence" SP iff $M \in \mathrm{Mod}[\mathrm{SP}]$. The satisfaction condition follows from the semantics of **translate** and the satisfaction condition for INS.

4.3. (*No*) *Data Constraints*

Our specification-building operations do not provide the possibility to explicitly require initially or freeness. This is in contrast to languages like Clear and Look but in accord with ASL and CIP-L. We could easily add such an operation (see below). That we do not do this is really just a matter of taste, strongly supported by some results on the power of

specification methods which indicate that no expressive power is lost by excluding such an operation, as long as some operation like our **minimal** is available and specifications may be built in a hierarchical fashion (in the standard algebraic framework, for monomorphic specifications—see [BBTW 81] for details). We could also set forth the (somewhat demagogic) argument that after all one can get exactly the same effect by including axioms like data constraints [GB 84a] in the underlying institution.

An operation which imposes freeness requirements (like **data** as defined below) has the technical disadvantage that it is not monotonic with respect to inclusion of model classes, unlike all the specification-building operations introduced above. This does not apply to the use of data constraints as sentences in basic specifications. Monotonicity turns out to be crucial both in defining recursive parameterised specifications (Section 7) and in composing implementation steps (Section 8).

It is worth noting that data constraints were originally introduced under the rather serious restriction that the underlying institution be liberal, which essentially excludes axioms more powerful than infinitary Horn formulae (see [MM 84, Mak 85, Tar 84] for an analysis of this problem in the standard case, and [Tar 85, 86] for its generalisation to an arbitrary institution). The device of *duplex institutions* allows one to relax this restriction in such a way that it applies only to that part of the institution which is actually used in data constraints (see [GB 84a] for all the details). The construction below shows that formally even this is not necessary:

For any specification SP and signature morphism $\sigma: \Sigma \to \mathrm{Sig}[\mathrm{SP}]$ we could define **data** SP **over** σ as a specification having the semantics

$$\mathrm{Sig}[\textbf{data SP over } \sigma] = \mathrm{Sig}[\mathrm{SP}]$$

$$\mathrm{Mod}[\textbf{data SP over } \sigma] = \{M \in |\textbf{Mod}(\mathrm{Sig}[\mathrm{SP}])| \mid \text{for any } M' \in \mathrm{Mod}[\mathrm{SP}] \text{ and } \Sigma\text{-morphism } f: M|_\sigma \to M'|_\sigma \text{ there exists a unique } \mathrm{Sig}[\mathrm{SP}]\text{-morphism } f^*: M \to M' \text{ such that } f^*|_\sigma = f\}.$$

Technical digression. We show that this would essentially give the effect of the data constraints of [GB 84a]. Let us recall the relevant definitions from [GB 84a] first.

By a *theory* we mean any basic specification $\langle \Sigma, \Phi \rangle$ such that Φ is a set of Σ-sentences closed under logical consequence; i.e., Φ satisfies the equality

$$\Phi = \{\varphi \in \textbf{Sen}(\Sigma) \mid M \models \varphi \text{ for all } M \in \mathrm{Mod}[\langle \Sigma, \Phi \rangle]\}.$$

By a *theory morphism* from $T1 = \langle \Sigma 1, \Phi 1 \rangle$ to $T2 = \langle \Sigma 2, \Phi 2 \rangle$ (where $T1$ and $T2$ are theories) we mean any signature morphism $\sigma: \Sigma 1 \to \Sigma 2$ such

that $\sigma(\Phi 1) \subseteq \Phi 2$. According to the satisfaction condition, this means exactly that the σ-reduct functor $-|_\sigma$ maps models of $T2$ to models of $T1$, and so for every theory morphism $\sigma: T1 \to T2$ there is an associated σ-reduct functor $-|_\sigma: \mathrm{Mod}[T2] \to \mathrm{Mod}[T1]$ (we identify classes of models with full subcategories of the category of models over the given signature).

An institution is called *liberal* if for every theory morphism $\sigma: T1 \to T2$, the σ-reduct functor $-|_\sigma: \mathrm{Mod}[T2] \to \mathrm{Mod}[T1]$ has a left adjoint, which we denote by $\mathbf{F}_\sigma: \mathrm{Mod}[T1] \to \mathrm{Mod}[T2]$ (with unit η^σ and counit ε^σ). Then, $M2 \in \mathrm{Mod}[T2]$ is *σ-free* if it is naturally isomorphic to the free model over $M2|_\sigma$, i.e., if the counit morphism $\varepsilon^\sigma_{M2|_\sigma}: \mathbf{F}_\sigma(M2|_\sigma) \to M2$ is an isomorphism.

Finally, a *Σ-data constraint* is a pair $\langle \sigma: T1 \to T2,\ \theta: \Sigma 2 \to \Sigma \rangle$ where $\sigma: T1 \to T2$ is a theory morphism, $\Sigma 2$ is the signature of $T2$, and $\theta: \Sigma 2 \to \Sigma$ is a signature morphism. A Σ-model M *satisfies* the data constraint $\langle \sigma: T1 \to T2,\ \theta: \Sigma 2 \to \Sigma \rangle$ if $M|_\theta$ is a model of $T2$ and is σ-free.

With these preliminary definitions, we can state the following key lemma:

LEMMA. *For any theory morphism $\sigma: T1 \to T2$, a $Sig[T2]$-model $M2$ is σ-free iff $M2 \in Mod$*[**data** $T2$ **over** σ].

Proof Sketch. The "if" part easily follows from the fact that any two free objects are naturally isomorphic (see, e.g., [AM 75]).

For the "only if" part note that if $\varepsilon^\sigma_{M2|_\sigma}: \mathbf{F}_\sigma(M2|_\sigma) \to M2$ is an isomorphism then $(\varepsilon^\sigma_{M2|_\sigma})^{-1}: M2 \to \mathbf{F}_\sigma(M2|_\sigma)$ is the unique (iso)morphism such that $(\varepsilon^\sigma_{M2|_\sigma})^{-1}|_\sigma = \eta^\sigma_{M2|_\sigma}$—the rest follows easily. ∎

COROLLARY. *A data constraint $\langle \sigma: T1 \to T2,\ \theta: \Sigma 2 \to \Sigma \rangle$ is equivalent to (has the same class of models as) the specification* **translate** (**data** $T2$ **over** σ) **by** θ.

The above construction indicates that data constraints may be defined without the assumption that the underlying institution is liberal. However, what happens then is that a data constraint may be inconsistent (have no model) even though the theories involved are consistent (have models). On the other hand, it must be realised that this may happen even if the underlying institution is liberal. Moreover, a specification which includes a number of data constraints may be inconsistent anyway. (End of technical digression)

5. Two Standard Cases

The definitions of the specification-building operations we gave in the last section were so general that they may be difficult to understand. We now consider what the operations do in two familiar contexts—the

institution FOEQ of first-order logic with equality as the only predicate symbol, and an institution PFOEQ of partial first-order logic—and compare them with operations in existing specification languages.

We define the institution FOEQ as follows:

— $\mathbf{Sign}_{\mathrm{FOEQ}}$ is **AlgSig** (i.e., $\mathbf{Sign}_{\mathrm{GEQ}}$, the category of algebraic signatures and their morphisms).

— $\mathbf{Mod}_{\mathrm{FOEQ}}$ is $\mathbf{Mod}_{\mathrm{GEQ}}$ (i.e., for any algebraic signature Σ, $\mathbf{Mod}_{\mathrm{FOEQ}}(\Sigma)$ is the category of Σ-algebras and for any algebraic signature morphism $\sigma: \Sigma \to \Sigma'$, $\mathbf{Mod}_{\mathrm{FOEQ}}(\sigma)$ is the σ-reduct functor from $\mathbf{Mod}_{\mathrm{FOEQ}}(\Sigma')$ to $\mathbf{Mod}_{\mathrm{FOEQ}}(\Sigma)$).

— For any algebraic signature Σ, $\mathbf{Sen}_{\mathrm{FOEQ}}(\Sigma)$ is the set of closed first-order formulae with operation symbols from Σ and the equality as the only predicate symbol; for any algebraic signature morphism $\sigma: \Sigma \to \Sigma'$, $\mathbf{Sen}_{\mathrm{FOEQ}}(\sigma)$ is the translation of Σ-formulae to Σ'-formulae defined in the natural way.

— The satisfaction relation is determined by the standard notion of satisfaction of first-order sentences.

This clearly forms an institution (details in [GB 84a]). Moreover, our assumptions that the category of signatures is finitely cocomplete and that $\mathbf{Mod}_{\mathrm{FOEQ}}$ translates finite colimits in $\mathbf{Sign}_{\mathrm{FOEQ}}$ to limits in **Cat** obviously hold here too; in fact, these parts of the institution are exactly the same as those in GEQ.

In the following we analyse the specification-building operations defined in Section 4 in the framework of the above institution of first-order logic.

There is hardly anything to be said about basic specifications. All specification languages provide a syntactic tool for listing a set of axioms over a given signature, although usually they differ in which formulae are acceptable. First-order equational axioms are relatively powerful compared with, e.g., equations in [GTW 76] or universal Horn axioms in [EKTWW 80].

In our examples we use a syntax corresponding to that of Clear:

Bool =	**sorts**	bool
	opns	true, false: → bool
		not: bool → bool
		or: bool, bool → bool
	axioms	∀x: bool. true or x = true
		not(true) = false
		not(false) = true
		∀x: bool. false or x = x
		∀x: bool. x = true ∨ x = false.

(Of course, *or* and ∨ are formally not the same here.)

The union operation differs from the corresponding operation in other specification languages (e.g., + in Clear or ASL) in that it works only for specifications over the same signature, and so it provides no direct way for putting together specifications over different signatures. To do this, we must use union together with the **translate** operation, which introduces new sorts and operation symbols to a specification (and renames old ones).

The sum of two specifications (as defined in ASL) may now be expressed as

$$\mathrm{SP} + \mathrm{SP}' =_{\mathrm{def}} (\textbf{translate SP by } \iota) \cup (\textbf{translate SP}' \textbf{ by } \iota'),$$

where $\iota: \mathrm{Sig}[\mathrm{SP}] \to \mathrm{Sig}[\mathrm{SP}] \cup \mathrm{Sig}[\mathrm{SP}']$ and $\iota': \mathrm{Sig}[\mathrm{SP}'] \to \mathrm{Sig}[\mathrm{SP}] \cup \mathrm{Sig}[\mathrm{SP}']$ are the inclusions of Sig[SP] and Sig[SP'], respectively, into their set-theoretic union $\mathrm{Sig}[\mathrm{SP}] \cup \mathrm{Sig}[\mathrm{SP}']$. To avoid unintended confusion of symbols with the same names in Sig[SP] and Sig[SP'], instead of using the inclusions ι and ι' we could use injections which rename the common symbols as required (as in Clear). This obviously generalises to arbitrary (not necessary finite) sums.

An operation similar to **enrich** in Clear (identical when there are no symbol clashes) may be defined in terms of the sum and basic specification operations:

$$\textbf{enrich SP by sorts S opns } \Omega \textbf{ axioms } \Phi$$
$$=_{\mathrm{def}} \mathrm{SP} + \langle \langle \mathrm{sorts}(\mathrm{SP}) \cup S, \mathrm{opns}(\mathrm{SP}) \cup \Omega \rangle, \Phi \rangle,$$

where $\mathrm{Sig}[\mathrm{SP}] = \langle \mathrm{sorts}(\mathrm{SP}), \mathrm{opns}(\mathrm{SP}) \rangle$.

Note that the **translate** operation corresponds directly to the TRA operator of [EWT 83].

The **derive** operation is, in a sense, dual to **translate**. It may be used to rename and to hide some of the sorts and operation symbols of a specification. It is exactly the same as **derive** in ASL [SW 83, short version only] and corresponds directly to the reflection (REF) operator in [EWT 83]. The intention is the same as that of **derive** in Clear, but the meaning is slightly different as mentioned in Section 4. The difference is highlighted by the following example, using the equational variant of Clear:

$$\mathrm{SP} = \textbf{derive from } \mathrm{Nat} \textbf{ by } \sigma: \Sigma \to \mathrm{Sig}[\mathrm{Nat}],$$

where Nat specifies (using, e.g., the **minimal** operation—see below) the standard model of the natural numbers with a single sort *nat* and operations $0: \to \mathrm{nat}$ and $\mathrm{succ}: \mathrm{nat} \to \mathrm{nat}$, Σ is $\mathrm{Sig}[\mathrm{Nat}] - \{0: \to \mathit{nat}\}$, and σ is the inclusion. According to our semantics of **derive**, Mod[SP] will be the class of Σ-algebras which look just like the standard model of the natural numbers but with the operation $0: \to \mathit{nat}$ missing. According to Clear's semantics of **derive** (making the appropriate slight changes to the syntax of

the example), any Σ-algebra at all will be a model of SP. This result seems inappropriate. The problem is that the semantics of Clear maps specifications to theories. It is impossible to give a theory-level semantics of our **derive** which works for examples like this one because not all the properties of models of the derived specification are expressible using just Σ-sentences.

Note that the collection of models of **derive from** SP **by** σ need not be closed under isomorphism even if Mod[SP] is. This phenomenon occurs when σ is not injective on sorts. When for two sorts s and s', $\sigma(s) = \sigma(s')$, **derive from** SP **by** σ requires the carriers of sorts s and s' to be identical rather than only isomorphic. (See below for some further discussion on this point.)

The **minimal** operation restricts the models of a specification SP to only those algebras which contain (to within isomorphism) no proper subalgebra which is a model of SP with the same σ-reduct. In particular, in the institution of first-order logic the definition of **minimal** as given in Section 4 states that if an algebra A is a model of the specification **minimal** SP **wrt** σ then A is a model of SP and whenever B is a model of SP which is a subalgebra of A such that $B|_\sigma = A|_\sigma$, then $A = B$. Moreover, if Mod[SP] is closed under isomorphism then the converse of this implication is true as well. In general, however, this need not be the case. Consider the counterexample

$$\text{SP} = \textbf{enrich (derive from } \langle \langle \textbf{sorts t} \rangle, \varnothing \rangle \textbf{ by } \sigma) \textbf{ by ops a}: \to \text{s, b}: \to \text{s}',$$

where σ: $\langle \textbf{sorts } s, s' \rangle \to \langle \textbf{sorts } t \rangle$. Now, a Sig[SP]-algebra A is a model of SP if and only if $|A|_s = |A|_{s'}$. Hence, for example, $A \in \text{Mod[SP]}$ where $|A|_s = \{0, 1\} = |A|_{s'}$ with $a_A = 0 \in |A|_s$ and $b_A = 1 \in |A|_{s'}$. Now, A contains no proper subalgebra which is a model of SP. Note, however, that there is a model B of SP which is isomorphic to a proper subalgebra of A ($|B|_s = \{*\} = |B|_{s'}$) so A is not minimal in Mod[SP] w.r.t. the inclusion of the empty signature into Sig[SP].

The **minimal** operation is similar to the GEN operator of [EWT 83] rather than to the **reachable** operation of ASL [SW 83] or the use of finitely generated algebras in CIP-L [Bau 81]. In fact, minimality does not guarantee reachability (and hence, for example, the induction principle need not hold in minimal algebras) although reachability does imply minimality

$$\begin{aligned} \text{NN} = \textbf{sorts} \quad & \text{nat} \\ \textbf{opns} \quad & \text{zero}: \to \text{nat} \\ & \text{succ}: \text{nat} \to \text{nat} \\ \textbf{axioms} \quad & \exists \text{x}: \text{nat}.\ \text{succ}(\text{x}) = \text{x} \end{aligned}$$

$$\text{Nat}_\omega = \textbf{minimal NN wrt } \iota_{\text{Sig[NN]}}$$

(we accept the convention that for any signature Σ, ι_Σ denotes the unique signature morphism from the initial signature to Σ; in particular, here $\iota_{\mathrm{Sig[NN]}}$ is the inclusion of the empty signature into Sig[NN]). Models of NN contain (up to isomorphism) either a finite segment $\{0, ..., n\}$ of natural numbers $\mathbb{N}$ with $\mathrm{succ}(n) = n$ and an arbitrary unreachable part or else $\mathbb{N}$ together with an arbitrary unreachable part containing at least one element x such that $\mathrm{succ}(x) = x$. The only models of Nat_ω are (up to isomorphism) finite segments $\{0, ..., n\}$ of $\mathbb{N}$ with $\mathrm{succ}(n) = n$ and all elements reachable, or else $\mathbb{N}$ together with exactly one unreachable element ω such that $\mathrm{succ}(\omega) = \omega$.

An operation which is like **reachable** in ASL [SW 83] may be defined in terms of **minimal** as

$$\textbf{reachable SP wrt } \sigma =_{\mathrm{def}} \mathrm{SP} + \textbf{minimal } \langle \mathrm{Sig[SP]}, \varnothing \rangle \textbf{ wrt } \sigma.$$

The **reachable** operation of ASL is in fact a special case of the above,

$$\textbf{reachable SP on } S =_{\mathrm{def}} \textbf{reachable SP wrt } \iota,$$

where ι is the inclusion of the signature $\langle \mathrm{sorts(SP)} - \mathrm{S}, \varnothing \rangle$ into Sig[SP]. For example,

$$\mathrm{Nat\text{-}seg} = \textbf{reachable NN wrt } \iota_{\mathrm{Sig[NN]}} = \textbf{reachable NN on } \{\mathrm{nat}\}.$$

Now, the only models of Nat-seg are (up to isomorphism) finite segments $\{0, ..., n\}$ of $\mathbb{N}$ with $\mathrm{succ}(n) = n$ and all elements reachable.

The **iso close** operation closes the collection of models of a specification under isomorphism. The only situation in which the collection of models of a specification may not be closed under isomorphism already is when the specification contains a use of **derive from** ... **by** σ where σ is not injective on sorts. It would be easy to "fix" **derive** by changing the definition so that the result is automatically closed under isomorphism (this was the alternative adopted in ASL [SW 83, long version]). Another possible "solution," which turns out to yield exactly the same expressive power, is to restrict **derive** by allowing only signature morphisms which are injective on sorts. We prefer, however, to adopt neither solution, retaining both **derive** (as it is defined now) and **iso close**. This is consistent with our policy of providing operations which are as elementary as possible. It also leaves open the possibility of specifying collections of models which are not closed under isomorphism; despite the well-known arguments that closure under isomorphism is natural, we feel that there is no harm in providing such flexibility.

The **derive** operation allows one to hide some of the sorts and operation symbols of a specification. This also causes some of the properties of its

models to be hidden, since they cannot be expressed using the remaining operations. However, this is not real abstraction yet since the structure induced by the hidden operations remains. To do real abstraction we can pick out a set of properties we would like to preserve and then use the **abstract** operation.

The properties we would like to preserve must be expressed as sentences of the underlying institution. However, to deal properly with unreachable elements of models (dubbed "junk" in [BG 81]) we must use open formulae rather than (closed) sentences. Why not just forbid junk? Although unreachable elements seem to be of no consequence, there is an example (Infinite-Set) in [SW 83] which shows how an unreachable element in a model of SP can become reachable and useful in **enrich** SP **by opns** .... Furthermore, junk naturally arises when we "forget" operations using **derive**, which corresponds to the situation where an algebra which is reachable when viewed from a low level becomes non-reachable when viewed from a higher level of abstraction.

The most natural way one may view **abstract** in the institution of first-order logic is, we think, the following.

Given a Σ-specification SP, identify the set Φ of properties which are to be preserved under abstraction. These properties must be expressed as $\Sigma(X)$-sentences, where X is a set of variables necessary to name unreachable elements. The abstraction of SP with respect to Φ is given by the specification **abstract** SP **wrt** Φ **via** ι where $\iota: \Sigma \to \Sigma(X)$ is the algebraic signature inclusion. This specifies (roughly speaking) the collection of Σ-algebras which satisfy the same formulae of Φ as models of SP. More formally, a Σ-algebra A satisfies **abstract** SP **wrt** Φ **via** ι if and only if there is a Σ-algebra B which satisfies SP and which has the property that for any valuation $v_A: X \to |A|$ there exists a valuation $v_B: X \to |B|$ such that for any formula $\varphi \in \Phi$, φ holds in A under the valuation v_A if and only if φ holds in B under the valuation v_B (and vice versa: for any $v_B: X \to |B|$ there exists $v_A: X \to |A|$ such that...).

To make this clearer, let us consider a simple example which does not make use of open formulae,

$$\text{Nat} = \textbf{minimal} \langle \Sigma, \{\forall x\text{: nat. } 0 \neq \text{succ}(x), \quad \forall x, y\text{: nat. } (\text{succ}(x) = \text{succ}(y) \Rightarrow x = y)\} \rangle \textbf{ wrt } \iota_\Sigma,$$

where

$$\begin{array}{lll} \Sigma = \textbf{sorts} & \text{nat} \\ \quad \textbf{opns} & 0: \to \text{nat} \\ & \text{succ: nat} \to \text{nat} \end{array}$$

$$\begin{aligned}\text{Nat-even} = \textbf{enrich Bool} + \text{Nat } \textbf{by opns} \quad & \text{even: nat} \to \text{bool} \\ \textbf{axioms} \quad & \text{even}(0) = \text{true} \\ & \text{even}(\text{succ}(0)) = \text{false} \\ & \forall x\text{:nat. even}(\text{succ}(\text{succ}(x))) \\ & \qquad = \text{even}(x).\end{aligned}$$

All models of Nat are isomorphic to the standard model of the natural numbers. (Note that for this specification minimality guarantees reachability.) Each model of Nat-even is the combination of a model of Nat with a model of Bool (see above) with an extra operation *even*. We can abstract from Nat-even preserving only the properties of booleans and the behaviour of *even* as follows,

$$\text{Nat-mod} = \textbf{abstract Nat-even wrt } \Phi_{\text{bool}} \textbf{ via } \text{id}_{\Sigma},$$

where $\text{id}_{\Sigma}: \Sigma \to \Sigma$ is the identity signature morphism, and

$$\begin{aligned}\Phi_{\text{bool}} = & \{t = t' \mid t, t' \text{ are } \Sigma\text{-terms of sort bool}\} \\ & \cup \{\forall x\text{: bool. } x = \text{true} \vee x = \text{false}\}.\end{aligned}$$

All models of Nat-mod are isomorphic either to the natural numbers modulo n, for some $n \in \{2, 4, 6, \ldots\}$, or to $\mathbb{N}$ itself with arbitrary junk of sort *nat* in both cases. The sentence $\forall x$: *bool.* $x = true \vee x = false$ in Φ_{bool} forces all models of Nat-mod to have only reachable elements of sort *bool*; if it were removed from Φ_{bool} then models of Nat-mod would contain arbitrary junk of sort *bool*.

We could achieve the same result using observations which are open formulae as follows,

$$\text{Nat-mod}' = \textbf{abstract Nat-even wrt } \Phi'_{\text{bool}} \textbf{ via } \iota,$$

where X is a set of variables with $X_{\text{nat}} = \varnothing$ and $X_{\text{bool}} \neq \varnothing$, $\iota: \Sigma \to \Sigma(X)$ is the algebraic signature inclusion, and

$$\Phi'_{\text{bool}} = \{t = t' \mid t, t' \text{ are } \Sigma\text{-terms of sort bool with variables } X\}.$$

The models of Nat-mod′ are exactly those of Nat-mod. Note that all models of Nat-mod′ have only reachable elements of sort *bool* since Φ'_{bool} contains the formulae $x = true$ and $x = false$ for some $x \in X_{\text{bool}}$. Note that we were able to achieve the same effect above using only ground observations (sentences with no free variables) only because the (reachable) carriers of sort *bool* in all models of Nat-even are finite. In other examples this is not the case. We give a much more detailed analysis of the properties of observational abstraction in [ST 87].

The idea of comparing algebras w.r.t. a set of formulae also appeared in

[Pep 83]. The difference is that there only closed formulae were considered. In ASL [SW 83] there is also a specification-building operation called **abstract** which corresponds to the special case of observational abstraction where observations are required to be equations. We generalised this operation to the framework of an arbitrary institution in [ST 84]. The approach of the present paper although originally derived from [ST 84] is technically different (and, we believe, simpler and more elegant).

The **abstract** operation may be used to relax the interpretation of a specification to all models which are behaviourally equivalent to a model of the specification (this is called *behavioural abstraction* in ASL [SW 83]—see that paper for examples).

Suppose that Σ is an algebraic signature and IN and OUT are subsets of the sorts of Σ. Consider all computations which take input from sorts IN and give output in sorts OUT; this set of computations corresponds to the set $|T_\Sigma(X_{IN})|_{OUT}$ of Σ-terms of sorts OUT with variables of sorts IN. Two algebras are equivalent in our sense with respect to the set $EQ(|T_\Sigma(X_{IN})|_{OUT})$ of equations between terms of the same sort in $|T_\Sigma(X_{IN})|_{OUT}$ if they are behaviourally equivalent; that is they have matching input/output relations. This covers the notions of behavioural equivalence with respect to a single set OBS of *observable* sorts which appear in the literature. For example, in [Rei 81] and [GM 82] we have IN = sorts(Σ), OUT = OBS; in [Sch 82, SW 83, GM 83] IN = OUT = OBS; and in [GGM 76, BM 81, Kam 83] IN = $\varnothing$ and OUT = OBS. In the case where IN = $\varnothing$ we have no control over the unreachable elements of observable sorts whatsoever. To express the obserations which are needed to guarantee the preservation of carriers of observable sorts we need free variables as in the case where IN = OBS.

The **abstract** operation usually does not appear explicitly in specification languages (the only exception we know about is ASL); instead, it is somehow included in the notion of the implementation of one specification by another. The inclusion of **abstract** as an explicit specification-building operation allows us to use a very simple and elegant definition of implementation (see Section 8 for a few details). On the other hand, **abstract** makes inference more complex because it is not monotone (at the level of theories) in the sense that things true in SP need not be true in **abstract** SP **wrt** ... (see Section 6).

A good test for the general definitions in Section 4 is to consider their instantiation in several different institutions. In the rest of this section we discuss the result of instantiating in an institution of partial first-order logic.

Let $\Sigma = \langle S, \Omega \rangle$ be an algebraic signature. A *partial Σ-algebra* is just like

a (total) Σ-algebra except that its operations may be partial. A (*weak*) Σ-*homomorphism* from a partial Σ-algebra A to a partial Σ-algebra B, $h: A \to B$, is a family of (total) functions $\{h_s\}_{s \in S}$ where $h_s: |A|_s \to |B|_s$ such that for any $f: s1, ..., sn \to s$ and $a_1 \in |A|_{s1}, ..., a_n \in |A|_{sn}$

$$f_A(a_1, ..., a_n) \text{ defined} \Rightarrow f_B(h_{s1}(a_1), ..., h_{sn}(a_n)) \text{ defined} \quad \text{and}$$
$$h_s(f_A(a_1, ..., a_n)) = f_B(h_{s1}(a_1), ..., h_{sn}(a_n))$$

([BrW 82] would call this a *total* Σ-homomorphism). If moreover h satisfies the condition

$$f_B(h_{s1}(a_1), ..., h_{sn}(a_n)) \text{ defined} \Rightarrow f_A(a_1, ..., a_n) \text{ defined}$$

then h is called a *strong* Σ-*homomorphism.*

The category of partial Σ-algebras $\mathbf{PAlg}(\Sigma)$ has partial Σ-algebras as objects and (weak) Σ-homomorphisms as morphisms; the composition of homomorphisms is the composition of their corresponding components as functions. (This obviously forms a category.)

The definition of the σ-*reduct* functor $-|_\sigma: \mathbf{PAlg}(\Sigma') \to \mathbf{PAlg}(\Sigma)$, where $\sigma: \Sigma \to \Sigma'$ is an algebraic signature morphism, is exactly the same as that in the total case; see Section 2.

A *partial first-order* Σ-*sentence* is a closed first-order formula built from Σ-terms using the logical connectives $\neg$, $\wedge$, $\vee$, and $\Rightarrow$, the quantifiers $\forall$ and $\exists$, and the atomic formulae $D_s(t)$ and $t = t'$ (strong equality [BrW 82]) for each sort s in Σ and terms $t, t' \in |T_\Sigma(X)|_s$ (i.e., t, t' are Σ-terms of sort s with variables X).

A partial Σ-algebra A *satisfies* an atomic formula $D_s(t)$ under a valuation $v: X \to |A|$, written $A \models_v D_s(t)$, iff the value of t in A under v is defined (we omit the definition of the value of a term in a partial algebra under a valuation; see [Bur 82, Rei 87] for details). A satisfies an atomic formula $t = t'$ (where $t, t' \in |T_\Sigma(X)|_s$ for some sort s in Σ) under a (total) valuation $v: X \to |A|$, written $A \models_v t = t'$, iff

$A \not\models_v D_s(t)$ and $A \not\models_v D_s(t')$, or

$A \models_v D_s(t)$ and $A \models_v D_s(t')$ and the values of t and t' in A under v are the same.

Satisfaction of (closed) partial first-order Σ-sentences is defined as usual, but note that $\forall$ and $\exists$ quantify only over defined values.

The institution PFOEQ of partial first-order logic is then defined as follows:

— $\mathbf{Sign}_{\mathrm{PFOEQ}}$ is **AlgSig**.

— For an algebraic signature Σ, $\mathbf{Mod}_{\mathrm{PFOEQ}}(\Sigma)$ is $\mathbf{PAlg}(\Sigma)$; for an algebraic signature morphism $\sigma: \Sigma \to \Sigma'$, $\mathbf{Mod}_{\mathrm{PFOEQ}}(\sigma)$ is the σ-reduct functor $-|_\sigma: \mathbf{PAlg}(\Sigma') \to \mathbf{PAlg}(\Sigma)$.

— For an algebraic signature Σ, $\mathbf{Sen}_{\mathrm{PFOEQ}}(\Sigma)$ is the set of partial first-order Σ-sentences as defined above; for an algebraic signature morphism $\sigma: \Sigma \to \Sigma'$, $\mathbf{Sen}_{\mathrm{PFOEQ}}(\sigma)$ is the translation of Σ-sentences to Σ'-sentences, defined in the obvious way.

— For an algebraic signature Σ, $\models_{\Sigma,\mathrm{PFOEQ}}$ is the satisfaction relation as defined above.

This forms an institution; the satisfaction condition follows from the fact that FOEQ is an institution and that definedness of terms is preserved under change of signature. Moreover, $\mathbf{Sign}_{\mathrm{PFOEQ}}$ is finitely cocomplete (as mentioned in Section 2) and $\mathbf{Mod}_{\mathrm{PFOEQ}}$ translates finite colimits in $\mathbf{Sign}_{\mathrm{PFOEQ}}$ to limits in **Cat**.

The result of instantiating the general definitions of Section 4 in PFOEQ gives a set of operations which in some respects resemble those in the early version of ASL described in [Wir 82] defined in the context of partial algebras (call this language "partial ASL," but note that it is significantly different from the ASL described in [SW 83]). One difference, however, is that in partial ASL the collection of models of any specification was closed under renaming of sorts and operations; i.e., if $\mathrm{Sig}[\mathrm{SP}] = \Sigma$ and $\Sigma \cong \Sigma'$, then $\mathrm{Mod}[\mathrm{SP}]$ contains partial Σ'-algebras as well as partial Σ-algebras. This feature could be obtained by changing the definition of $\mathbf{Mod}_{\mathrm{PFOEQ}}$ and $\models_{\Sigma,\mathrm{PFOEQ}}$ but we prefer to omit it.

The earlier comments regarding basic specifications and the union, **translate**, **derive**, and **iso close** operations (and how to define + in terms of $\cup$ and **translate**) in the context of the institution FOEQ apply without change here.

As expected, **minimal** SP **wrt** σ gives the least-defined and smallest (w.r.t. σ) models of SP. This operation may be used to express the operation **mdef** of partial ASL, albeit in a rather unsatisfactory way:

$$\mathbf{mdef}\ \mathrm{SP} \underset{\mathrm{def}}{=} (\mathrm{SP} \cup \mathbf{minimal}\ \langle \mathrm{Sig}[\mathrm{SP}], D \rangle\ \mathbf{wrt}\ \iota_{\mathrm{Sig}[\mathrm{SP}]}) + \mathrm{Bool},$$

where $D =_{\mathrm{def}} \{D_s(t) \mid t \in |T_{\mathrm{Sig}[\mathrm{SP}]}|_s,\ s \in \mathrm{sorts}(\mathrm{Sig}[\mathrm{SP}]),$ and $M \models D_s(t)$ for all $M \in \mathrm{Mod}[\mathrm{SP}]\}$ and Bool is a specification of the booleans including the axiom *true* $\neq$ *false*.

Abstract works similarly as in FOEQ. The use of **abstract** for behavioural abstraction is slightly different though, since the properties to be preserved must include definedness of the results of "observable" computations. If Σ is an algebraic signature and IN, OUT are subsets of the sorts of Σ as before, behavioural equivalence in the context of partial algebras may be defined as observational equivalence with respect to the set of formulae $\mathrm{EQ}(|T_\Sigma(X_{\mathrm{IN}})|_{\mathrm{OUT}}) \cup \{D_s(t) \mid t \in |T_\Sigma(X_{\mathrm{IN}})|_s$ for $s \in \mathrm{OUT}\}$. Partial ASL includes no operation similar to **abstract**.

It is instructive to note how a small change to the institution PFOEQ may affect the operations. For example, we can consider an institution which is exactly the same as PFOEQ except that for any algebraic signature Σ, we consider only *strong* Σ-homomorphisms between partial Σ-algebras. The **minimal** operation has a completely different meaning in this institution: **minimal** SP **wrt** σ gives the smallest (w.r.t. σ) models of SP in each class of equally defined models. The meanings of the other operations remain unchanged.

6. Proofs in Structured Specifications

In the framework of an arbitrary institution INS, for any signature Σ each class of Σ-models K determines a theory $\mathrm{Th}(K) = \{\varphi \in \mathbf{Sen}(\Sigma) \mid M \models \varphi$ for all $M \in K\}$, i.e., the set of all Σ-sentences which are true in every model belonging to K (note however that the class of models satisfying $\mathrm{Th}(K)$ may properly include K). So every Σ-specification SP determines the set of its logical consequences, the set $\mathrm{Th}(\mathrm{SP}) = \mathrm{Th}(\mathrm{Mod}[\mathrm{SP}])$ of all Σ-sentences which hold in all its models. These are exactly the properties of the specified object expressible in the given institution on which a user is allowed to rely.

In the above, we said nothing about how to effectively determine if a property (sentence) follows from a specification. Our basic notion is the satisfaction relation and model-theoretic (rather than proof-theoretic) consequence. All the same, for practical purposes it is necessary to have some effective (=computational) way of proving that a sentence is a consequence of a specification, i.e., a *proof system.*

Notation. $\mathrm{SP} \models \varphi$ means that the sentence φ *holds* in all models of SP ($\varphi \in \mathrm{Th}(\mathrm{SP})$). $\mathrm{SP} \vdash \varphi$ means that φ is *provable* from SP in a given proof system.

Any useful proof system must be *sound*; that is $\mathrm{SP} \vdash \varphi$ must imply $\mathrm{SP} \models \varphi$ (we must only be able to prove things which are true). Another important property which a proof system may have is *completeness*, i.e., $\mathrm{SP} \models \varphi$ implies $\mathrm{SP} \vdash \varphi$ (we can prove all the true things). Unfortunately, for every practical specification approach no sound and complete effective proof system can exist; more precisely, this holds for every specification approach which is powerful enough to specify the natural numbers—see [MS 85] for a review of this problem in the context of equational specifications. So we must be content with a proof system which is sound but not complete. The same situation occurs in program verification; there is no (Cook-) complete Hoare-like proof system for any programming language with a sufficiently rich control structure [Cla 79].

Of course, we cannot expect to be able to construct a satisfactory (i.e., "complete enough") proof system which is entirely independent of the institution in use. We must assume that we are given some (sound) proof system for the underlying institution, that is a proof system which allows us to deduce sentences from sets of sentences (basic specifications). This amounts to a proof system for any specification language where specification-building operations are defined at the level of presentations. However, this does not imply that such a semantics is required for doing theorem proving. It is possible to extend the proof system for the underlying institution to a proof system for the specification language. What we must do is to devise an inference rule for every specification-building operation which allows facts about a compound specification to be deduced from facts about its components [SB 83] in a way which does not depend on the particular properties of the underlying institution. This approach allows us to use the structure of the specification to direct the search for a proof, which is necessary to control the amount of information present in large specifications. An additional benefit is that the resulting proof will reflect the structure of the specification.

Let us consider our specification-building operations one by one.

INFERENCE RULE (union): *For any signature Σ and family $\{SP_i\}_{i\in I}$ of Σ-specifications, for any $i \in I$ and any Σ-sentence φ,*

$$SP_i \vdash \varphi \Rightarrow \bigcup_{i\in I} SP_i \vdash \varphi.$$

FACT. *The family of inference rules above is sound; i.e.,* $\mathrm{Th}(\bigcup_{i\in I} SP_i) \supseteq \bigcup_{i\in I} \mathrm{Th}(SP_i)$, *where the second* $\bigcup$ *denotes the set-theoretic union of sets of Σ-sentences.*

Proof. It is enough to show that $\mathrm{Mod}[\bigcup_{i\in I} SP_i] \subseteq \mathrm{Mod}[\bigcup_{i\in I} \mathrm{Th}(SP_i)]$, which follows directly from the definition. ▮

Moreover, in the case where the specifications $\{SP_i\}_{i\in I}$ are basic specifications (in fact, it is sufficient to require that $\mathrm{Mod}[\mathrm{Th}(SP_i)] = \mathrm{Mod}[SP_i]$ for $i \in I$) the inclusion of model classes opposite to the one given in the proof holds as well, and so $\mathrm{Th}(\bigcup_{i\in I} SP_i) \subseteq \mathrm{Th}(\bigcup_{i\in I} \mathrm{Th}(SP_i))$. This shows that the above family of inference rules is, in a sense, complete. This is the best completeness result we can hope for, since after all an inference rule cannot "see" those properties of the component specifications which are not expressible in the underlying institution (and, as mentioned before, we cannot do the job hidden in the use of Th above which is the responsibility of the proof system for the underlying institution).

INFERENCE RULE (**translate**): *For any signature morphism* $\sigma: \Sigma \to \Sigma'$, *$\Sigma$-specification SP, and Σ-sentence* φ,

$$\mathrm{SP} \vdash \varphi \Rightarrow \textbf{translate SP by } \sigma \vdash \sigma(\varphi).$$

FACT. *The above inference rule is sound; i.e.,* $\mathrm{Th}(\textbf{translate SP by } \sigma) \supseteq \sigma(\mathrm{Th}(\mathrm{SP}))$.

Proof. It is enough to show that $\mathrm{Mod}[\textbf{translate SP by } \sigma] \subseteq \mathrm{Mod}[\sigma(\mathrm{Th}(\mathrm{SP}))]$, which follows directly from the definition of **translate** and the satisfaction condition for the underlying institution. ∎

Again, in the case where $\mathrm{Mod}[\mathrm{Th}(\mathrm{SP})] = \mathrm{Mod}[\mathrm{SP}]$ the inclusion of model classes opposite to the one given in the proof holds as well, and so $\mathrm{Th}(\textbf{translate SP by } \sigma) \subseteq \mathrm{Th}(\sigma(\mathrm{Th}(\mathrm{SP})))$.

INFERENCE RULE (**derive**). *For any signature morphism* $\sigma: \Sigma \to \Sigma'$, *$\Sigma'$-specification SP', and Σ-sentence* φ,

$$\mathrm{SP}' \vdash \sigma(\varphi) \Rightarrow \textbf{derive from SP' by } \sigma \vdash \varphi.$$

FACT. *The above inference rule is sound; i.e.,* $\mathrm{Th}(\textbf{derive from SP' by } \sigma) \supseteq \sigma^{-1}(\mathrm{Th}(\mathrm{SP}'))$.

Proof. It is enough to note that $\mathrm{Mod}[\textbf{derive from SP' by } \sigma] \subseteq \mathrm{Mod}[\sigma^{-1}(\mathrm{Th}(\mathrm{SP}'))]$, which follows directly from the definition of **derive** and the satisfaction condition. ∎

It follows directly from the satisfaction condition and the definition of **derive** that for any Σ-sentence φ we have $\sigma(\varphi) \in \mathrm{Th}(\mathrm{SP}')$ iff $\varphi \in \mathrm{Th}(\textbf{derive from SP' by } \sigma)$; i.e., $\sigma^{-1}(\mathrm{Th}(\mathrm{SP}')) = \mathrm{Th}(\textbf{derive from SP' by } \sigma)$. Incidentally, this implies that $\sigma^{-1}(\mathrm{Th}(\mathrm{SP}'))$ is closed under consequence. However, even in the case where $\mathrm{Mod}[\mathrm{Th}(\mathrm{SP})] = \mathrm{Mod}[\mathrm{SP}]$, the inclusion of model classes opposite to the one given in the proof above need not hold.

INFERENCE RULE (**minimal**). *For any signature morphism* $\sigma: \Sigma' \to \Sigma$, *$\Sigma$-specification SP, and Σ-sentence* φ,

$$\mathrm{SP} \vdash \varphi \Rightarrow \textbf{minimal SP wrt } \sigma \vdash \varphi.$$

The above inference rule is obviously sound. However, it does not reflect the restriction which the **minimal** operation imposes. The most typical use of the **minimal** operation is to restrict interpretation of a specification SP to its reachable models (this is exactly the case when Mod[SP] is closed under submodels). In the standard algebraic framework, this allows us to use an appropriate form of structural induction as the inference rule

associated with this specification-building operation. Note however that structural induction is itself typically not complete (see [MS 85] for discussion of this problem in the framework of equational logic).

For the **iso close** operation, note that in general there need not be any connection between the truth of sentences and the morphism structure of the categories of models. However, in practically all the institutions we are dealing with, the truth of sentences is preserved under isomorphism (that is, isomorphic models are elementarily equivalent). If this is the case, then for any Σ-specification SP, $\mathrm{Th}(\mathrm{SP}) = \mathrm{Th}(\textbf{iso close}\ \mathrm{SP})$, which gives an obvious (and trivial) inference rule for **iso close**.

The inference rule for **abstract** is going to be a bit more complicated, partly because in contrast to the other operations, **abstract** (intentionally) does not preserve the truth of sentences; i.e.,

$$\mathrm{SP} \models \varphi \not\Rightarrow \textbf{abstract}\ \mathrm{SP}\ \textbf{wrt}\ \ldots\ \textbf{via}\ \ldots \models \varphi.$$

In order to give the inference rule, we need the following notation. For any signature morphism $\theta\colon \Sigma \to \Sigma'$ and set $\Phi' \subseteq \mathrm{Sen}(\Sigma')$ of open Σ-formulae, we define $\mathrm{Cl}(\Phi')$ to be the least set of Σ-sentences closed (insofar as the institution allows) under conjunction, negation, and equivalence, and including the sentences $\forall\langle\psi, \theta\rangle$ and $\exists\langle\psi, \theta\rangle$ for every ψ in the least set of Σ'-sentences containing Φ' and closed under conjunction, negation, and equivalence (we use universal and existential quantification of open formulae in an arbitrary institution as introduced in Section 3).

INFERENCE RULE (**abstract**). *For any Σ-specification SP, signature morphism $\theta\colon \Sigma \to \Sigma'$, set $\Phi' \subseteq Sen(\Sigma')$ of open Σ-formulae, and Σ-sentence φ,*

$$SP \vdash \varphi \text{ and } \varphi \in \mathrm{Cl}(\Phi') \ \Rightarrow\ \textbf{abstract}\ \mathrm{SP}\ \textbf{wrt}\ \Phi'\ \textbf{via}\ \theta \vdash \varphi.$$

FACT. *The above inference rule is sound; i.e.,* $\mathrm{Th}(\textbf{abstract}\ \mathrm{SP}\ \textbf{wrt}\ \Phi'\ \textbf{via}\ \theta) \supseteq \mathrm{Th}(\mathrm{SP}) \cap \mathrm{Cl}(\Phi')$.

The proof uses the definition of **abstract** and of the satisfaction of quantified formulae in an arbitrary institution. For the proof, for a more detailed discussion of the Cl operation and of the completeness of this inference rule, and for an example of its use, see [ST 87].

Note that for any sound proof system $\vdash$, if SP is a Σ-specification and φ is a Σ-sentence such that $\mathrm{SP} \vdash \varphi$ then

$$\mathrm{Mod}[\mathrm{SP}] = \mathrm{Mod}[\mathrm{SP} \cup \langle\Sigma, \varphi\rangle].$$

This suggests the possibility of incrementally combining a specification with its logical consequences as they are discovered. This would be useful from a practical point of view in order to avoid repeating the same proof twice, and is reminiscent of the Z specification language [ASM 79] in which

specifications can contain theorems in addition to axioms. This also suggests that it may be interesting to introduce and investigate some notion of a theory which reflects the structure of the specification from which the theory is derived (our thanks to Rod Burstall for this observation). In such a *structured theory* each sentence (theorem) would be attached to the smallest subspecification of which it is a consequence.

7. Parameterised Specifications

Besides providing a certain collection of predefined specification-building operations, it is desirable to allow a user to define his own specification-building operations, i.e., to provide a mechanism for constructing *parameterised specifications.* There are different approaches to parameterised specifications; the ones which seem most natural in our framework are those which treat a parameterised specification as a function from specifications to specifications as in, e.g., Clear [BG 80], Look [ETLZ 82], or ASL. A typical parameterised specification is Stack, which when applied to a specification of the elements which are to be "stacked" yields a specification of stacks of those elements. As with procedures in programming languages, a parameterised specification consists of two parts: a *formal parameter* providing a "skeleton" which any *actual parameter* must match, and a description of how an actual parameter is manipulated to form the result.

The way to deal with parameterised specifications which is most widely accepted in the literature on algebraic specifications (e.g., [BG 80, Ehr 79]) is based on the pushout construction in the category of specifications. (We switch here for a moment to the usual algebraic framework where specifications are just theories; specification morphisms are signature morphisms which presere axioms.) In this approach, a parameterised specification P is a specification morphism $P: \mathrm{SP}_{\mathrm{par}} \to \mathrm{SP}_{\mathrm{res}}$ from the formal parameter specification to the result specification (usually P is assumed to be an inclusion). To apply such a parameterised specification to an actual parameter specification $\mathrm{SP}_{\mathrm{act}}$ we must provide another specification morphism which "fits" $\mathrm{SP}_{\mathrm{act}}$-models into $\mathrm{SP}_{\mathrm{par}}$-models, $\sigma: \mathrm{SP}_{\mathrm{par}} \to \mathrm{SP}_{\mathrm{act}}$ (recall that then the reduct functor $-|_\sigma$ takes $\mathrm{SP}_{\mathrm{act}}$-models to $\mathrm{SP}_{\mathrm{par}}$-models). The result of applying P to $\mathrm{SP}_{\mathrm{act}}$ using σ is the specification $P(\mathrm{SP}_{\mathrm{act}}[\sigma])$ which is defined (up to isomorphism) as the pushout object of P and σ, that is

$$\begin{array}{ccc} \mathrm{SP}_{\mathrm{act}} & \xrightarrow{P'} & P(\mathrm{SP}_{\mathrm{act}}[\sigma]) \\ \uparrow{\scriptstyle \sigma} & & \uparrow{\scriptstyle \sigma'} \\ \mathrm{SP}_{\mathrm{par}} & \xrightarrow[P]{} & \mathrm{SP}_{\mathrm{res}} \end{array}$$

is a pushout in the category of specifications.

This may be generalised to the framework of an arbitrary institution INS. For readers interested in the technical details, here is a sketch of how this may be done.

Technical digression. First, we must define the category of specifications. For any Σ-specification SP and Σ'-specification SP', a specification morphism $\sigma: \mathrm{SP} \to \mathrm{SP}'$ is a signature morphism $\sigma: \Sigma \to \Sigma'$ such that for any $M' \in \mathrm{Mod}[\mathrm{SP}']$, $M'|_\sigma \in \mathrm{Mod}[\mathrm{SP}]$. This obviously yields a category of specifications $\mathbf{Spec}_{\mathrm{INS}}$ (with identities and composition defined as in the category of signatures). The category of theories $\mathbf{Th}_{\mathrm{INS}}$ as defined in [GB 84a] is a full subcategory of $\mathbf{Spec}_{\mathrm{INS}}$. Note that the function Sig: $|\mathbf{Spec}_{\mathrm{INS}}| \to |\mathbf{Sign}_{\mathrm{INS}}|$ extends in a natural way to a functor **Sig**.

THEOREM. *The functor* $\mathbf{Sig}: \mathbf{Spec}_{\mathrm{INS}} \to \mathbf{Sign}_{\mathrm{INS}}$ *reflects colimits.*

Proof Sketch. Let D be a diagram in $\mathbf{Spec}_{\mathrm{INS}}$ with nodes SP_i for $i \in I$. Let Σ with injections $\sigma_i: \mathrm{Sig}[\mathrm{SP}_i] \to \Sigma$ be a colimit of D; **Sig** (in $\mathbf{Sign}_{\mathrm{INS}}$). Let SP be the Σ-specification $\bigcup_{i \in I}$ **translate** SP_i **by** σ_i. It is easy to check that SP with injections $\sigma_i: \mathrm{SP}_i \to \mathrm{SP}$ for $i \in I$ is a colimit of D. ▮

Note that the theorem in [GB 84a] that the functor $\mathbf{Sig}: \mathbf{Th}_{\mathrm{INS}} \to \mathbf{Sign}_{\mathrm{INS}}$ reflects colimits follows from the above construction since **translate** can be defined at the level of presentations and theories, as mentioned in Section 4.1.

Now, since we assume that $\mathbf{Sign}_{\mathrm{INS}}$ is finitely cocomplete, it follows that $\mathbf{Spec}_{\mathrm{INS}}$ is finitely cocomplete, and hence the pushout used in the above construction always exists.

Incidentally, it is worth noting that the function Mod: $|\mathbf{Spec}_{\mathrm{INS}}| \to |\mathbf{Cat}|$ (mapping any Σ-specification SP to the full subcategory of the category of Σ-models with objects Mod[SP]) may also be extended in a natural way to a (contravariant) functor $\mathbf{Mod}: \mathbf{Spec}_{\mathrm{INS}} \to \mathbf{Cat}^{\mathrm{op}}$. It follows from the above construction of colimits in $\mathbf{Spec}_{\mathrm{INS}}$ that **Mod** is finitely cocontinuous, i.e., maps initial objects and pushouts in $\mathbf{Spec}_{\mathrm{INS}}$ to terminal objects and pullbacks, respectively, in **Cat** (the proof uses the satisfaction condition and the assumption that the model functor $\mathbf{Mod}_{\mathrm{INS}}: \mathbf{Sign}_{\mathrm{INS}} \to \mathbf{Cat}^{\mathrm{op}}$ of the underlying institution is finitely cocontinuous). This generalises the condition (mentioned in Section 2) imposed on the logical framework in [EWT 83]. (End of technical digression)

The proof in the above technical digression shows that the result of applying P to $\mathrm{SP}_{\mathrm{act}}$ using the fitting morphism σ (in the pushout approach) is easily expressible using our specification-building operations as

$$(\textbf{translate } \mathrm{SP}_{\mathrm{act}} \textbf{ by } P') \cup (\textbf{translate } \mathrm{SP}_{\mathrm{res}} \textbf{ by } \sigma'),$$

where P' and σ' are as above.

We would like to adopt a more elementary view of parameterised specifications, closer to parameterisation mechanisms in programming languages. Our treatment is based on the mechanism of macro-expansion (β-conversion in the λ-calculus). A similar but again more complicated approach was pursued in ASL [SW 83] (for other versions of this approach see [Wir 82, 83]). Semantically, any parameterised specification can be viewed as a function taking any specification over the given parameter signature Σ_{par} to a specification over the result signature Σ_{res}.

Formally, a parameterised specification is just a λ-expression $\lambda X\colon \Sigma_{\text{par}}.\text{SP}_{\text{res}}$ where X is an identifier, Σ_{par} is the parameter signature, and SP_{res} is a Σ_{res}-specification built using specification-building operations which may involve X as a variable denoting a Σ_{par}-specification. For any Σ_{par}-specification SP, $(\lambda X\colon \Sigma_{\text{par}}.\text{SP}_{\text{res}})(\text{SP})$ is a specification where

$$\text{Sig}[(\lambda X\colon \Sigma_{\text{par}}.\text{SP}_{\text{res}})(\text{SP})] = \Sigma_{\text{res}}$$

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

(we adopt the usual λ-calculus convention that $E[v/x]$ denotes the result of substituting v for x in E). Another way of handling the semantics of parameterised specifications is to explicitly regard a parameterised specification $\lambda X\colon \Sigma_{\text{par}}.\text{SP}_{\text{res}}$ as denoting a function $[\![\lambda X\colon \Sigma_{\text{par}}.\text{SP}_{\text{res}}]\!]$ from classes of Σ_{par}-models to classes of Σ_{res}-models defined in the obvious way. Then $\text{Mod}[(\lambda X\colon \Sigma_{\text{par}}.\Sigma_{\text{res}})(\text{SP})] = [\![\lambda X\colon \Sigma_{\text{par}}.\Sigma_{\text{res}}]\!](\text{Mod}[\text{SP}])$. It is easy to check that the function $[\![\lambda X\colon \Sigma_{\text{par}}.\text{SP}_{\text{res}}]\!]$ is monotone w.r.t. inclusion of classes of models (since all the specification-building operations we provide are).

Note that we require an exact match between the parameter signature Σ_{par} and the signature of the actual parameter specification SP. This means that any fitting which is necessary must be done explicitly before application. However, we require only that the signature of SP fits Σ_{par} as in [Sch 82]; in contrast to the pushout-based approach there is no restriction on the class of models of SP, since the result of application is well-defined for any Σ_{par}-specification.

ASL permits specifications to be defined recursively. We can do the same here. Whenever we have a parameterised specification $\lambda X\colon \Sigma.\text{SP}_{\text{res}}$ with the same parameter and result signature Σ, its denotation $[\![\lambda X\colon \Sigma.\text{SP}_{\text{res}}]\!]$ is a monotone function from classes of Σ-models to classes of Σ-models. This function always has a greatest (w.r.t. inclusion of classes of models) fixed point. We can thus introduce a specification-building operation **fix** as follows: **fix** $\lambda X\colon \Sigma.\text{SP}_{\text{res}}$ is a Σ-specification with the greatest fixed point of $[\![\lambda X\colon \Sigma.\text{SP}_{\text{res}}]\!]$ as its class of models. Some examples of the use of this mechanism are given in [SW 83, Wir 83].

In the above we described how to handle parameterised specifications

having a single parameter. In order to handle multiple parameters we can either combine them into a single big parameter (this is the way the semantics of Clear works) or slightly extend our treatment of parameterised specifications to permit parameterised specifications which yield parameterised specifications as a result (that is, to allow $\mathrm{SP}_{\mathrm{res}}$ to itself be a parameterised specification). Then the multiple parameters can be handled by "currying": instead of $\lambda X{:}\,\Sigma 1,\ Y{:}\,\Sigma 2.\mathrm{SP}_{\mathrm{res}}$ we write $\lambda X{:}\,\Sigma 1.(\lambda Y{:}\,\Sigma 2.\mathrm{SP}_{\mathrm{res}})$. We can pursue the latter solution even further and permit an arbitrary hierarchy of higher-order parameterised specifications by allowing both the arguments and the results of parameterised specifications to be parameterised specifications of an arbitrary complexity.

Note that in the above we have viewed a parameterised specification as a specification-building operation and so we have applied the function $[\![\lambda X{:}\,\Sigma_{\mathrm{par}}.\mathrm{SP}_{\mathrm{res}}]\!]$ to (the class of models of) the actual parameter specification "as a whole." An alternative is to apply it "pointwise" to each of the models of the actual parameter and then form the union of the resulting model classes. In general this gives a different result,

$$[\![\lambda X{:}\,\Sigma_{\mathrm{par}}.\mathrm{SP}_{\mathrm{res}}]\!](\mathrm{Mod}[\mathrm{SP}]) \supseteq \bigcup_{M \in \mathrm{Mod}[\mathrm{SP}]} [\![\lambda X{:}\,\Sigma_{\mathrm{par}}.\mathrm{SP}_{\mathrm{res}}]\!](\{M\}),$$

where the inclusion may be proper, for example, if $\mathrm{SP}_{\mathrm{res}}$ contains two occurrences of X. The right-hand side of this inclusion suggests another possible way to define the semantics of parameterised specifications which may even be more appropriate in a context where we are interested in building models rather than specifications.

8. Concluding Remarks

The work presented in this paper is aimed toward application in the systematic development of programs from specifications. We have not yet discussed the development process itself. The programming discipline of stepwise refinement suggests that a program be evolved by working gradually via a series of successively lower-level refinements of the specification toward a specification which is so low level that it can be regarded as a program. For example, the specification

$$\mathrm{reverse}(\mathrm{nil}) = \mathrm{nil}$$
$$\mathrm{reverse}(\mathrm{cons}(a, l)) = \mathrm{append}(\mathrm{reverse}(l),\mathrm{cons}(a,\mathrm{nil}))$$

is an executable program in Standard ML [Mil 84]. The stepwise refinement approach guarantees the correctness of the resulting program, provided that each refinement step can be proved correct. A formalisation

of this approach requires a precise definition of the concept of refinement, i.e., of the *implementation* of one specification by another.

In programming practice, proceeding from a specification to a program means making a series of design decisions. These will include decisions concerning the concrete representation of abstractly defined data types, decisions about how to compute abstractly specified functions (choice of algorithm), and decisions which select between the various possibilities which the high-level specification leaves open. The following very simple formal notion of implementation (independent from the particular institution in use) captures this idea: a specification SP is implemented by another specification SP′, written $SP \rightsquigarrow SP'$, if SP′ incorporates more design decisions than SP; i.e., any model of SP′ is a model of SP (SP and SP′ are required to have the same signature). We can adopt this simple notion if we have an operation like observational abstraction available (see [SW 83, ST 87] for more discussion on this point).

This notion of implementation can be extended to give a notion of the implementation of parameterised specifications: P is implemented by P', written $P \rightsquigarrow P'$, if P and P' have the same parameter signature Σ and for all Σ-specifications SP, $P(SP) \rightsquigarrow P'(SP)$.

An important issue for any notion of implementation is whether implementations can be composed *vertically* and *horizontally* [GB 80]. Implementations can be vertically composed if the implementation relation is transitive ($SP \rightsquigarrow SP'$ and $SP' \rightsquigarrow SP''$ implies $SP \rightsquigarrow SP''$) and they can be horizontally composed if the specification-building operations preserve implementations ($P \rightsquigarrow P'$ and $SP \rightsquigarrow SP'$ implies $P(SP) \rightsquigarrow P'(SP')$). The above notion of implementation has both these properties, since all our specification-building operations are monotonic (with respect to inclusion of model classes). These two properties allow large structured specifications to be refined in a gradual and modular fashion. All of the individual small specifications which make up a large specification can be separately refined in several stages to give a collection of lower-level specifications (this should be relatively easy because of their small size). When the low-level specifications are put back together, the result is guaranteed to be an implementation of the original specification. Note that other more complicated notions of implementation ([EKMP 82], just to take one example) do not compose vertically or horizontally in general.

We have not studied in detail the interactions between the specification-building operations we have defined. It is obvious, however, that they satisfy some non-trivial laws. For example, it is possible to prove identities such as

$$\textbf{translate (abstract SP wrt } \Phi' \textbf{ via } \theta) \textbf{ by } \sigma$$
$$= \textbf{abstract (translate SP by } \sigma) \textbf{ wrt } \sigma'(\Phi') \textbf{ via } \theta',$$

where $\sigma': \Sigma' \to \Sigma 1'$ and $\theta': \Sigma 1 \to \Sigma 1'$ are the pushout in **Sign** of $\sigma: \Sigma \to \Sigma 1$ and $\theta: \Sigma \to \Sigma'$ (as in Section 3). A "library" of laws of this kind could be used as a basis for program development. For a more detailed analysis and examples of other laws which hold between our specification-building operations in the standard algebraic framework, see [SW 83].

In this paper we defined and analysed a set of primitive and general specification-building operations which when instantiated in any institution provide a powerful but low-level tool for specification. We tested the institution-based general definitions of these operations by examining the result of instantiating them in two different ways: in an institution of total first-order equational logic and in an institution of partial first-order logic (Section 5). When we originally formulated the definitions we also considered the result of instantiating them in two other institutions, an error institution based on [GDLE 82] and an institution of continuous algebras based on [ANR 85], cf. [TW 86].

The question of whether the definitions we have given are really general naturally arises; maybe there is some institution which we have not considered in which the operations we have defined work in an unexpected way. Indeed, whenever one generalises on the basis of a small collection of examples one must choose between all the generalisations which are different in general but which coincide in the particular examples one has at hand. For, example, in the definition of the **minimal** operation, to represent the concrete notion of injective homomorphisms we used just monomorphisms rather than, say, equalisers or extremal monomorphisms (or more generally we could parameterise our definition by an image factorisation system as in [Tar 85]). All of these possibilities work equally well in each of our example institutions. We can try to test our generalisations by comparing them with other available general definitions. So for example we can show (see [Tar 85]) that—under certain not very restrictive conditions—minimal corresponds to "generated" as defined in [GB 84a] (note however that the definition of [GB 84a] works only in liberal institutions, and this is a strong restriction).

Another natural question concerns our decision to allow the specification of collections of models which are not closed under isomorphism and our careful treatment of models containing unreachable elements. We chose this course because we cannot see any really compelling reason, either pragmatic or technical, for assuming that all useful collections of models are closed under isomorphism or that only reachable models are worth considering. On the other hand, we also know of no compelling reason why these assumptions (especially the former) are unreasonable. By leaving the choice to the specifier (or to the designer of a high-level specification language which builds upon our kernel operations) we provide the freedom to explore all possibilities without unnecessary restrictions.

Although the reader might have the impression that we have been carried away in our pursuit of generality, we tried to resist the urge to throw in unnecessary generalisations. So, for example, it is clear that **iso close** can be generalised to give an operation which can close under different classes of morphisms, and not just under isomorphism. This generalisation might even be useful; note that closure under (sources of) monomorphisms gives closure under subalgebras, and closure under (targets of) epimorphisms gives closure under quotients. We do not claim to offer every possible operation on collections of models, only a few interesting ones which we know are useful. This is also part of our justification for omitting an operation which restricts to the initial or final elements in a collection of models.

The theme which underlies all of the work presented in this paper is one of generality. Striving always to work at the most general level possible results in reusable theories and tools. We argued that it is best to avoid choosing any particular logical system on which to base a specification approach. Instead we parameterised our work by an *arbitrary* institution. We hope that we have convinced the reader that this is an appropriate level of generality on which to introduce and analyse concepts like specification and implementation, and tools like specification-building and theorem-proving formalisms.

ACKNOWLEDGMENTS

Our thanks to Rod Burstall for his support, encouragement, and some helpful comments, to Martin Wirsing for his work on ASL, to Gordon Plotkin for help with logic, to José Meseguer for suggesting some improvements to the presentation, and to Joseph Goguen for useful discussions. This research was supported by the Science and Engineering Research Council.

RECEIVED April 11, 1985; ACCEPTED August 4, 1986

REFERENCES

[ASM 79] ABRIAL, J. R., SCHUMAN, S. A., AND MEYER, B. (1979), "Specification language Z," Massachusetts Computer Associates Inc., Boston.

[ANR 85] ADAMEK, J., NELSON, E., AND REITERMAN, J. (1985), A Birkhoff variety theorem for continuous algebras, *Algebra Universalis* **20**, 328–350.

[AM 75] ARBIB, M. A., AND MANES, E. G. (1975), "Arrows, Structures and Functors: The Categorical Imperative," Academic Press, New York.

[Bar 74] BARWISE, K. J. (1974), Axioms for abstract model theory, *Ann. of Math. Logic* **7**, 221–265.

[Bau 81] BAUER, F. L., *et al.* (the CIP Language Group) (1981), "Report on a Wide

Spectrum Language for Program Specification and Development," Report TUM-I8104, Technische Univ. München.

[BR 83] BENECKE, K., AND REICHEL, H. (1983), Equational partiality, *Algebra Universalis* **16**, 219–232.

[BBTW 81] BERGSTRA, J. A., BROY, M., TUCKER, J. V., AND WIRSING, M. (1981), On the power of algebraic specifications, *in* "Proc. 10th Intl. Symp. on Mathematical Foundations of Computer Science, Strbske Pleso, Czechoslovakia," pp. 193–204, Lecture Notes in Computer Science Vol. 118, Springer-Verlag, Berlin/New York.

[BM 81] BERGSTRA, J. A., AND MEYER, J. J. (1981), I/O computable data structures, *SIGPLAN Notices* **16**(4), 27–32.

[BW 85] BLOOM, S. L., AND WAGNER, E. G. (1985), Many-sorted theories and their algebras with some applications to data types, *in* "Algebraic Methods in Semantics" (M. Nivat and J. C. Reynolds, Eds.), Cambridge Univ. Press.

[BrW 82] BROY, M., AND WIRSING, M. (1982), Partial abstract types, *Acta Inform.* **18**, 47–64.

[Bur 82] BURMEISTER, P. (1982), Partial algebras—Survey of a unifying approach towards a two-valued model theory for partial algebras, *Algebra Universalis* **15**, 306–358.

[BG 80] BURSTALL, R. M., AND GOGUEN, J. A. (1980), The semantics of Clear, a specification language, *in* "Proc. of Advanced Course on Abstract Software Specifications, Copenhagen," pp. 292–332, Lecture Notes in Computer Science, Vol. 86, Springer-Verlag, Berlin/New York.

[BG 81] BURSTALL, R. M., AND GOGUEN, J. A. (1981), An informal introduction to specifications using Clear, *in* "The Correctness Problem in Computer Science" (R. S. Boyer and J. S. Moore, Eds.), pp. 185–213, Academic Press, New York.

[Cla 79] CLARKE, E. M. (1979), Programming language constructs for which it is impossible to obtain good Hoare axiom systems, *J. Assoc. Comput. Mach.* **26**(1), 129–147.

[Ehr 79] EHRICH, H.-D. (1979), "On the Theory of Specification, Implementation, and Parametrization of Abstract Data Types," Report 82, Univ. of Dortmund; also in *J. Assoc. Comput. Mach.* **29**(1), 206–227 (1982).

[EFH 83] EHRIG, H., FEY, W., AND HANSEN, H. (1983), "ACT ONE: An Algebraic Specification Language with Two Levels of Semantics, Report No. 83-03, Institut für Software und Theoretische Informatik, Technische Univ. Berlin.

[EKMP 82] EHRIG, H., KREOWSKI, H.-J., MAHR, B., AND PADAWITZ, P. (1982), Algebraic implementation of abstract data types, *Theoret. Comput. Sci.* **20**, 209–263.

[EKTWW 80] EHRIG, H., KREOWSKI, H.-J., THATCHER, J. W., WAGNER, E. G., AND WRIGHT, J. B. (1980), Parameterized data types in algebraic specification languages (short version), *in* "Proc. 7th Intl. Colloq. on Automata, Languages and Programming, Noordwijkerhout, Netherlands," Lecture Notes in Computer Science Vol. 85, Springer-Verlag, Berlin/New York.

[ETLZ 82] EHRIG, H., THATCHER, J. W., LUCAS, P., AND ZILLES, S. N. (1982), "Denotational and Initial Algebra Semantics of the Algebraic Specification Language Look," Draft report, IBM research.

[EWT 83] EHRIG, H., WAGNER, E. G., AND THATCHER, J. W. (1983), Algebraic specifications with generating constraints, *in* "Proc. 10th ICALP, Barcelona," pp. 188–202, Lecture Notes in Computer Science Vol. 154, Springer-Verlag, Berlin/New York.

[Gau 84] GAUDEL, M. C. (1984), "An introduction to PLUSS," Draft report, Université de Paris-Sud, Orsay.

[GGM 76] GIARRATANA, V., GIMONA, F., AND MONTANARI, U. (1976), Observability concepts in abstract data type specification, *in* "Proc. 5th Intl. Symp. on Mathematical Foundations of Computer Science, Gdansk," Lecture Notes in Computer Science Vol. 45, Springer-Verlag, Berlin/New York.

[GDLE 82] GOGOLLA, M., DROSTEN, K., LIPECK, U., AND EHRICH, H.-D. (1982), "Algebraic and Operational Semantics of Specifications Allowing Expectations and Errors," Fb. 140, Abteilung Informatik, Univ. of Dortmund; also in (1984), *Theoret. Comput. Sci.* **34**, 289–313.

[Gog 77] GOGUEN, J. A. (1977), Abstract errors for abstract data types, *in* "Proc. IFIP Working Conf. on the Formal Description of Programming Concepts, New Brunswick, NJ."

[Gog 78] GOGUEN, J. A. (1978), "Order Sorted Algebras: Expectations and Error Sorts, Coercions and Overloaded Operators," Semantics and Theory of Computation Report No. 14, Department of Computer Science, UCLA.

[GB 84a] GOGUEN, J. A., AND BURSTALL, R. M. (1984), Introducing institutions, *in* "Proc. Logics of Programming Workshop" (E. Clarke and D. Kozen, Eds.), Carnegie-Mellon University, pp. 221–256, Lecture Notes in Computer Science, Vol. 164, Springer-Verlag, Berlin/New York.

[GB 84b] GOGUEN, J. A., AND BURSTALL, R. M. (1984), Some fundamental algebraic tools for the semantics of computation. Part 1. Comma categories, colimits, signatures and theories, *Theoret. Comput. Sci.* **31**, 175–210.

[GM 82] GOGUEN, J. A., AND MESEGUER, J. (1982), Universal realization, persistent interconnection and implementation of abstract modules, *in* Proc. 9th ICALP, Aarhus, Denmark," pp. 265–281, Lecture Notes in Computer Science Vol. 140, Springer-Verlag, Berlin/New York.

[GM 83] GOGUEN, J. A., AND MESEGUER, J. (1983), "An Initiality Primer," Draft report, SRI International.

[GTW 76] GOGUEN, J. A., THATCHER, J. W., AND WAGNER, E. G. (1976), "An initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types," IBM research report RC 6487; also in (1978), "Current Trends in Programming Methodology. Vol. 4. Data Structuring" (R. T. Yeh, Ed.), pp. 80–149, Prentice-Hall, Englewood Cliffs, NJ.

[GH 83] GUTTAG, J. V., AND HORNING, J. J. (1983), "Preliminary Report on the Larch Shared Language," Report CSL-83-6, Computer Science Laboratory, Xerox PARC.

[Kam 83] KAMIN, S. (1983), Final data types and their specification, *TOPLAS* **5**(1), 97–121.

[Lis 81] LISKOV, B., ATKINSON, R., BLOOM, T., MOSS, E., SCHAFFERT, J. C., SCHEIFLER, R., AND SNYDER, A. (1981), "CLU Reference Manual," Lecture Notes in Computer Science Vol. 144, Springer-Verlag, Berlin/New York.

[LB 77] LISKOV, B. H., AND BERZINS, V. (1977), "An Appraisal of Program Specifications," Computation structures group memo 141–1, Laboratory for Computer Science, MIT.

[MacL 71] MACLANE, S. (1971), "Categories for the Working Mathematician," Springer-Verlag, Berlin/New York.

[MS 85] MACQUEEN, D. B., AND SANNELLA, D. T. (1985), Completeness of proof systems for equational specifications, *IEEE Trans. Software Engrg.* **SE-11**, 454–461.

[MM 84] MAHR, B., AND MAKOWSKY, J. A. (1984), Characterizing specification languages which admit initial semantics, *Theoret. Comput. Sci.* **31**, 49–60.

[Mak 85] MAKOWSKY, J. A. (1985), Why Horn formulas matter in computer science: Initial structures and generic examples, *in* "Proc. 10th Colloq. on Trees in Algebra and Programming, Joint Conf. on Theory and Practice of Software Development (TAPSOFT), Berlin," pp. 374–387, Lecture Notes in Computer Science, Springer-Verlag, Berlin/New York.

[Mil 84] MILNER, R. G. (1984), A proposal for Standard ML, *in* "Proc. 1984 ACM Symp. on LISP and Functional Programming, Austin, Texas."

[Moore 56] MOORE, E. F. (1956), Gedanken-experiments on sequential machines, "Automata Studies" (C. E. Shannon and J. McCarthy, Eds.), pp. 129–153, Princeton Univ. Press, Princeton.

[Pep 83] PEPPER, P. (1983), On the correctness of type transformations, talk at 2nd Workshop on Theory and Applications of Abstract Data Types, Passau.

[Rei 81] REICHEL, H. (1981), Behavioural equivalence—A unifying concept for initial and final specification methods, *in* "Proc. 3rd Hungarian Computer Science Conference, Budapest," pp. 27–39.

[Rei 87] REICHEL, H. (1987), Initial Computability, Algebraic Specifications, and Partial Algebras, Oxford Univ. Press.

[Sad 84] SADLER, M. (1984), "Mapping out specification," Position paper, "Workshop on Formal Aspects of Specification, Swindon."

[SB 83] SANNELLA, D. T., AND BURSTALL, R. M. (1983), Structured theories in LCF, *in* "Proc. 8th Colloq. on Trees in Algebra and Programming, L'Aquila, Italy," pp. 377–391, Lecture Notes in Computer Science Vol. 159, Springer-Verlag, Berlin/New York.

[ST 84] SANNELLA, D. T., AND TARLECKI, A. (1984), Building specifications in an arbitrary institution, *in* "Proc. Intl. Symposium on Semantics of Data Types, Sophia-Antipolis," pp. 337–356, Lecture Notes in Computer Science, Springer-Verlag, Berlin/New York.

[ST 85] SANNELLA, D. T., AND TARLECKI, A. (1985), Program specification and development in Standard ML, *in* "Proc 12th ACM Symp. on Principles of Programming Languages, New Orleans," pp. 67–77.

[ST 87] SANNELLA, D. T., AND TARLECKI, A. (1987), On observational equivalence and algebraic specification, *J. Comput. System Sci.* **34**, 150–178; extended abstract in (1985), "Proc. 10th Colloq. on Trees in Algebra and Programming, Joint Conf. on Theory and Practice of Software Development (TAPSOFT), Berlin," pp. 308–322, Lecture Notes in Computer Science Vol. 185, Springer-Verlag, Berlin/New York.

[SW 83] SANNELLA, D. T., AND WIRSING, M. (1983), "A Kernel Language for Algebraic Specification and Implementation," Report CSR-131-83, Department of Computer Science, Univ. of Edinburgh; extended abstract in (1983), "Proc. Intl. Conf. on Foundations of Computation Theory, Borgholm, Sweden," pp. 413–427, Lecture Notes in Computer Science Vol. 158, Springer-Verlag, Berlin/New York.

[Sch 82] SCHOETT, O. (1982), "A Theory of Program Modules, Their Specification and Implementation," Report CSR-155-83, Department of Computer Science, Univ. of Edinburgh.

[Suf 82] SUFRIN, B. (1982), Formal specification of a display-oriented text editor, *Sci. Comput. Programming* **1**, 157–202.

[Tar 84] TARLECKI, A. (1984), Free constructions in algebraic institutions, *in* "Proc. Intl. Symp. on Mathematical Foundations of Computer Science, Prague," pp. 526–534, Lecture Notes in Computer Science Vol. 176, Springer-Verlag, Berlin/New York; long version (1983), Report CSR-149-83, Department of Computer Science, Univ. of Edinburgh.

[Tar 85] TARLECKI, A. (1985), On the existence of free models in abstract algebraic institutions, *Theoret. Comput. Sci.* **37**, 269–304.

[Tar 86] TARLECKI, A. (1986), Quasi-varieties in abstract algebraic institutions, *J. Comput. Syst. Sci.* **33**, 333–360.

[TW 86] TARLECKI, A., AND WIRSING, M. (1986), Continuous abstract data types, *Fund. Inform.* **9**, 95–126; extended abstract in (1985), Continuous abstract data types—Basic machinery and results, *in* "Proc. Intl. Conf. on Fundamentals of Computation Theory, Cottbus, GDR," pp. 431–441, Lecture Notes in Computer Science Vol. 199, Springer-Verlag, Berlin/New York.

[Wand 79] WAND, M. (1979), Final algebra semantics and data type extensions, *J. Comput. System Sci.* **19**, 27–44.

[Wir 82] WIRSING, M. (1982), Structured algebraic specifications, *in* "Proc. AFCET Symp. on Mathematics for Computer Science, Paris," pp. 93–107.

[Wir 83] WIRSING, M. (1983), "Structured Algebraic Specifications: A Kernel Language," Habilitation thesis, Technische Univ. München.

[ZLT 82] ZILLES, S. N., LUCAS, P., AND THATCHER, J. W. (1982), "A Look at Algebraic Specifications," IBM research report RJ 3568.