ON OBSERVATIONAL EQUIVALENCE AND ALGEBRAIC SPECIFICATION

— Extended abstract[1] —

Donald Sannella and Andrzej Tarlecki[2]
Department of Computer Science
University of Edinburgh

Abstract
The properties of a simple and natural notion of observational equivalence of algebras and the corresponding specification-building operation (observational abstraction) are studied. We begin with a definition of observational equivalence which is adequate to handle reachable algebras only, and show how to extend it to cope with unreachable algebras and also how it may be generalised to make sense under an arbitrary institution. Behavioural equivalence is treated as an important special case of observational equivalence, and its central role in program development is shown by means of an example.

1 Introduction

Probably the most exciting potential application of formal specifications is to the formal development of programs by gradual refinement from a high-level specification to a low-level "program" or "executable specification" as in HOPE [BMS 80]. Each refinement step embodies some design decisions (such as choice of data representation) under the requirement that behaviour must be preserved. If each refinement step can be proved correct, then the program which results is guaranteed to satisfy the original specification.

This paper studies what is meant by "behaviour" in the context of algebraic specifications. Intuitively, the behaviour of a program is determined just by the answers which are obtained from computations the program may perform. 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

$|A|_{bunch}$ = the set of finite sets of natural numbers
$|B|_{bunch}$ = 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 $m \in add(a_1,\ldots,add(a_n,empty)\ldots)$). Note that A and B are not isomorphic.

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 this paper we deal

[1]The full version of this paper is available as report CSR-172-84, Department of Computer Science, University of Edinburgh.

[2]On leave from Institute of Computer Science, Polish Academy of Sciences, Warsaw.

with the more general situation in which observations may be arbitrary logical formulae. We discuss a notion of *observational equivalence* in which two algebras are observationally equivalent if they both give the same answers to any observation from a prespecified set.

Observational equivalence (or more specifically, 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 [ADJ 76] have suggested that "abstract" in "abstract data type" means "up to isomorphism"; we suggest that it really means "up to behavioural equivalence".

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]. 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 [ADJ 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. The kernel specification language ASL [SW 83] provides a specification-building operation **abstract** which when applied to a specification SP relaxes interpretation to all those algebras which are observationally equivalent to a model of SP with respect to the given set of "equational" observations. With a properly chosen set of observations, this gives *behavioural abstraction*.

Stepwise refinement

A formalisation of stepwise refinement requires a precise definition of the notion of refinement, i.e. of the *implementation* of one specification by a lower-level specification. In the context of a specification language which includes an operation like behavioural abstraction, it is possible to adopt a very simple definition of implementation (see section 5 for details). This notion of implementation has two very desirable properties (vertical and horizontal composability, see [GB 80]) which permit the development of programs from specifications in a gradual and modular fashion. An alternative approach which illustrates the same point is to use a definition of implementation which implicitly involves behavioural equivalence, as in [GM 82] and [Sch 83].

This paper establishes a number of basic definitions and results concerning observational equivalence in an attempt to provide a sound foundation for its application to problems such as those indicated above. We begin by treating in section 2 the case in which observations are logical formulae containing no free variables. We define observational equivalence of algebras and a specification-building operation (**abstract**) which performs observational abstraction and explore their basic properties. We generalise this material in two different dimensions; section 3 discusses observations which contain free variables (to handle "junk" in unreachable algebras without resorting to infinitary logic) and we also mention how the definitions can be generalised to make sense under an arbitrary logical system (or *institution* [GB 83]). Section 4 deals with the problem of proving theorems about structured specifications in the context of observational abstraction. Section 5 discusses behavioural equivalence as an important special case of observational equivalence. A simple notion of implementation is defined, and we demonstrate the role of behavioural equivalence in program development by carrying out one refinement step in the development of a fragment of an optimising compiler from its specification.

We assume that the reader is familiar with the basic algebraic notions presented in e.g. [ADJ 76] (cf. [BG 82]) as well as basic notions of logic as in e.g. [End 72] including some infinitary logic, see [Karp 64].

2 Observational equivalence: the ground case

What is an observation on an algebra? In the axiomatic framework, the most natural choice is to take logical formulae as observations; the result of an observation on an algebra is just the truth or falsity of the formula in the algebra. The kind of formulae we use dictates the kinds of observations we are allowed to make on algebras. On the other hand, the kinds of observations we want to make on algebras dictates the kind of formulae we need, that is the logic we should use.

For example, if we want only to examine results of computations, the natural choice is equations which allow us to compare the values of terms. Another natural choice is first-order predicate calculus which allows us to distinguish between e.g. closed and open intervals of rationals (the observation/formula $\forall x.\exists y.x<y$ yields true in the latter and false in the former). Another choice is an infinitary logic such as $L_{\omega_1\omega}$ which allows us to check e.g. reachability of algebras (that is, whether all elements of the algebra are values of ground terms). Note that the latter two kinds of observations are not computationally-based; they are at a more abstract level, i.e. they describe algebras rather than computations in algebras. Still another kind of formulae are necessary if we want to deal with e.g. problems of concurrency, but we disregard such issues in this paper.

For the moment, we do not want to commit ourselves to any particular logic, and so we leave the notion of "formula" undefined. (In fact, all our definitions work in an even more general setting.) The reader may feel more comfortable to imagine that we are talking about first-order logic.

We use the term "formula" rather than "sentence" to indicate the possible presence of free variables to name elements which are not values of ground terms ("junk"). Free variables introduce some complications which we postpone to the next section. We will assume for the remainder of this section that formulae contain no free variables; we call these *ground observations* or *sentences*. The following definition corresponds directly to the definition of *elementary equivalence* in [Pep 83].

Definition: Let Σ be a signature, Φ a set of Σ-sentences, and let A,B be Σ-algebras. A and B are *observationally equivalent with respect to* Φ, written $A\equiv_\Phi B$, if for any $\varphi\in\Phi$, $A\models\varphi$ iff $B\models\varphi$.

Easy facts

Fact 1: For any signature Σ and set Φ of Σ-sentences, $\equiv_\Phi$ is an equivalence relation on the class of Σ-algebras. □

Fact 2: For any signature Σ, sets Φ,Φ' of Σ-sentences, and Σ-algebras A,B, $\Phi\supseteq\Phi'$ and $A\equiv_\Phi B$ implies $A\equiv_{\Phi'} B$. □

Fact 3: For any signature Σ, family $\{\Phi_i\}_{i\in I}$ of sets of Σ-sentences, and Σ-algebras A,B, $A\equiv_{\Phi_i} B$ for all $i\in I$ implies $A\equiv_\Phi B$ where $\Phi = \bigcup_{i\in I}\Phi_i$. □

Two algebras are observationally equivalent wrt Φ if they satisfy exactly the same sentences of Φ. Note that this remains true if we consider not only the sentences of Φ but also their negations and conjunctions, possibly infinite or empty ($\bigwedge\phi$ is true). We can also add to Φ sentences equivalent to the ones already in Φ, and so everything which is definable in terms of negation and conjunction as well (disjunctions, implications etc.). For any set Φ of Σ-sentences, let $Cl(\Phi)$ denote the closure of Φ under negation, conjunction and equivalence, insofar as the logic in use allows.

Fact 4: $\equiv_\Phi = \equiv_{Cl(\Phi)}$ □

Note that this implies that the premise $\Phi\supseteq\Phi'$ in fact 2 may be replaced by the weaker condition $Cl(\Phi)\supseteq\Phi'$.

A *signature morphism* $\sigma:\Sigma\rightarrow\Sigma'$ is a renaming of the sorts and operations in Σ to those of Σ' which preserves the argument and result sorts of operations. This induces in a natural way a translation of Σ-terms to Σ'-terms and of Σ-sentences to Σ'-sentences; if φ is a Σ-sentence, then $\sigma(\varphi)$ denotes its translation to a Σ'-sentence. A signature morphism $\sigma:\Sigma\rightarrow\Sigma'$ also induces a *σ-reduct* functor translating any Σ'-algebra A' to a Σ-algebra $A'|_\sigma$. For the exact definitions of these notions see e.g. [ST 84, section 2]. These translations satisfy the following condition (see [GB 83]):

For any Σ-sentence φ and Σ'-algebra A', $A'|_\sigma\models\varphi$ iff $A'\models\sigma(\varphi)$ (Satisfaction condition)

This gives immediately the following fact:

Fact 5: For any signature morphism $\sigma:\Sigma\rightarrow\Sigma'$, set Φ of Σ-sentences and Σ'-algebras A',B', $A'\equiv_{\sigma(\Phi)}B'$ iff $A'|_\sigma\equiv_\Phi B'|_\sigma$ where $\sigma(\Phi) = \{\sigma(\varphi) \mid \varphi\in\Phi\}$. □

This says that observational equivalence is coherent with translation along signature morphisms. We can also show that observational equivalence is preserved under combination of "independent" algebras.

Let $\Sigma1$ and $\Sigma2$ be disjoint signatures and let $\Sigma1+\Sigma2$ be their (disjoint) union. For any $\Sigma1$-algebra $A1$ and $\Sigma2$-algebra $A2$, let $\langle A1,A2\rangle$ be the unique $\Sigma1+\Sigma2$-algebra such that $\langle A1,A2\rangle|_{\iota1}=A1$ and $\langle A1,A2\rangle|_{\iota2}=A2$, where $\iota1$ and $\iota2$ are the inclusions of $\Sigma1$ and $\Sigma2$ (respectively) into $\Sigma1+\Sigma2$. Note that all $\Sigma1+\Sigma2$-algebras are of this form.

Fact 6: For any disjoint signatures $\Sigma1,\Sigma2$, sets of $\Sigma1$-sentences $\Phi1$ and $\Sigma2$-sentences $\Phi2$, $\Sigma1$-algebras $A1,B1$ and $\Sigma2$-algebras $A2,B2$, $\langle A1,A2\rangle\equiv_{\iota1(\Phi1)\cup\iota2(\Phi2)}\langle B1,B2\rangle$ iff $A1\equiv_{\Phi1}B1$ and $A2\equiv_{\Phi2}B2$. □

A specification describes a collection of models of the same signature. To formalise this, for any specification SP let Sig[SP] denote its signature and Mod[SP] denote the class of its models, which are Sig[SP]-algebras. The notion of observational equivalence gives rise to a very powerful specification-building operation:

Definition: For any specification SP and set Φ of Sig[SP]-sentences

Sig[**abstract** SP **wrt** Φ] = Sig[SP]
Mod[**abstract** SP **wrt** Φ] = $\{ A \mid A\equiv_\Phi B$ for some $B\in$Mod[SP] $\}$

Informally, **abstract** SP **wrt** Φ is a specification which admits any model which is observationally equivalent to some model of SP. This provides a way of abstracting away from certain details of a specification (see [SW 83], [ST 84]).

<u>Easy facts</u>

Fact 7: For any specification SP and set Φ of Sig[SP]-sentences, Mod[SP] $\subseteq$ Mod[**abstract** SP **wrt** Φ]. □

Fact 8: For any specification SP and set Φ of Sig[SP]-sentences, Mod[**abstract** (**abstract** SP **wrt** Φ) **wrt** Φ] = Mod[**abstract** SP **wrt** Φ]. □

Fact 9: For any specification SP and sets Φ,Φ' of Sig[SP]-sentences, $\mathrm{Cl}(\Phi)\supseteq\Phi'$ implies Mod[**abstract** SP **wrt** Φ] $\subseteq$ Mod[**abstract** SP **wrt** Φ']. □

Fact 10: For any specifications SP,SP' such that Sig[SP]=Sig[SP'] and set Φ of Sig[SP]-sentences, Mod[SP]$\subseteq$Mod[SP'] implies Mod[**abstract** SP **wrt** Φ] $\subseteq$ Mod[**abstract** SP' **wrt** Φ]. □

Using the above facts we may derive simple identities which allow us to transform specifications involving **abstract**. For example:

Fact 11: For any specification SP and sets Φ,Φ' of Sig[SP]-sentences,

a. Mod[**abstract** SP **wrt** $\Phi\cup\Phi'$] $\subseteq$ Mod[**abstract** (**abstract** SP **wrt** Φ) **wrt** Φ']
$\subseteq$ Mod[**abstract** SP **wrt** $\mathrm{Cl}(\Phi)\cap\mathrm{Cl}(\Phi')$]

b. $\mathrm{Cl}(\Phi)\supseteq\Phi'$ implies
Mod[**abstract** SP **wrt** Φ'] = Mod[**abstract** (**abstract** SP **wrt** Φ) **wrt** Φ']
= Mod[**abstract** (**abstract** SP **wrt** Φ') **wrt** Φ] □

Note that the second equality in (b) need not hold if $\mathrm{Cl}(\Phi)\not\supseteq\Phi'$, and that the inclusions in (a) may be proper.

Every algebraic specification language provides an operation for specifying the class of models of a given signature which satisfy a given set of axioms, that is:

Definition: For any signature Σ and set Δ of Σ-sentences, $\langle\Sigma,\Delta\rangle$ is a *basic specification* and

$\mathrm{Sig}[\langle\Sigma,\Delta\rangle] = \Sigma$
$\mathrm{Mod}[\langle\Sigma,\Delta\rangle] = \{ A \mid A \text{ is a } \Sigma\text{-algebra and } A\models\Delta \}$

For any signature Σ and class K of Σ-algebras, let Th(K) denote the set of all Σ-sentences which hold in K. Note that $K\subseteq\mathrm{Mod}[\langle\Sigma,\mathrm{Th}(K)\rangle]$ but the converse inclusion is true only for classes K definable by basic specifications.

Fact 12: For any specification SP with Sig[SP]=Σ and set Φ of Σ-sentences, Mod[**abstract** SP **wrt** Φ] $\subseteq$ $\mathrm{Mod}[\langle\Sigma,\mathrm{Th}(\mathrm{Mod}[\mathrm{SP}])\cap\mathrm{Cl}(\Phi)\rangle]$. □

In the following, we try to further characterise how **abstract** works for classes of models definable by basic specifications. For any signature Σ and set Δ of Σ-sentences, let $\Delta^{\bullet} = \mathrm{Th}(\mathrm{Mod}[\langle\Sigma,\Delta\rangle])$ be the *closure of* Δ *under consequence.*

Fact 13: In first-order logic, for any signature Σ and sets Δ,Φ of Σ-sentences, Mod[**abstract** $\langle\Sigma,\Delta\rangle$ **wrt** Φ] = $\mathrm{Mod}[\langle\Sigma,\Delta^{\bullet}\cap\mathrm{Cl}(\Phi)\rangle]$.

Proof sketch (for $\supseteq$): For unsatisfiable Δ the containment holds; assume Δ has a model. Let $A\in\mathrm{Mod}[\langle\Sigma,\Delta^{\bullet}\cap\mathrm{Cl}(\Phi)\rangle]$, and $\Psi = \{\varphi\in\mathrm{Cl}(\Phi) \mid A\models\varphi\}$. Assume that $\Delta^{\bullet}\cup\Psi$ is not satisfiable. Then by the compactness theorem of first-order logic and since Ψ is closed under conjunction there is a $\psi\in\Psi$ such that $\Delta^{\bullet}\cup\{\psi\}$ has no model. Hence $\Delta^{\bullet}\models\neg\psi$, which, since $\Delta^{\bullet}$ is closed under logical consequence implies that $\neg\psi\in\Delta^{\bullet}$. Thus $\neg\psi\in\Delta^{\bullet}\cap\mathrm{Cl}(\Phi)$ and so $A\models\neg\psi$, which contradicts $\psi\in\Psi$. This proves that $\Delta^{\bullet}\cup\Psi$ has a model, say B. It is easy to check that $A\equiv_{\Phi}B$. □

An examination of the above proof shows that the fact holds for $L_{\omega_1\omega}$ as well as for first-order logic, and in fact for any infinitary logic (any logic which admits negation and conjunction of sets of formulae of any cardinality less than the cardinality in which the logic is compact).

Ground observations are powerful enough if we are only interested in reachable (subalgebras of) algebras and we do not want to distinguish between isomorphic algebras, provided that our logic is at least capable of expressing ground equations (i.e. equations between terms without variables).

Fact 14: For any signature Σ and Σ-algebras A,B, $A \equiv_{GEQ(\Sigma)} B$ iff A and B have isomorphic reachable subalgebras, where GEQ(Σ) is the set of all *ground* Σ-equations. □

3 Observational equivalence: the general case

In the last section we dealt with observational equivalence based on ground observations only (formally, on formulae without free variables). As fact 14 indicates, this is quite satisfactory when we restrict our considerations to reachable algebras. If we want to deal with algebras containing "junk" things become more complicated.

Why do we bother about non-reachable algebras? First, when dealing with parameterised specifications it is usual to consider examples in which some sorts have no generators at all, but where we are interested in algebras having the associated carriers non-empty. This is shown by standard examples such as Stack-of-X, where X is an arbitrary set. Second, when we view algebras from different levels of abstraction we view them with respect to different sets of operations. It is then natural that an algebra which is reachable at a certain level of abstraction becomes non-reachable when viewed from a higher level. A more technical but related point is that there is no natural definition of the specification-building operation **derive** [BG 80], [ST 84] (which can be used to "forget" operations) if models of the result are required to be reachable. Finally, there are examples [SW 83] in which unreachable elements can be useful in constructing specifications; an element which is unreachable at one stage of the construction can become reachable and useful at a later stage.

It should be noted that if the logic we are working in is sufficiently powerful then we can identify algebras up to isomorphism using only ground observations. According to Scott's Theorem [Scott 65] this may be achieved using $L_{\omega_1\omega}$ for countable algebras. Using an even more powerful logic the same may be done for arbitrary algebras.

Fact 15: For any signature Σ and Σ-algebra A, there is a Σ-sentence $\zeta(A)$ of $L_{\infty\infty}$ such that for any Σ-algebra B, $B \models \zeta(A)$ iff $A \cong B$.

Proof sketch: Assume for notational convenience that Σ has only one sort. Consider the formula $\zeta(A) =_{def} \exists |A|.(\bigwedge \mathscr{D}(A)\ \&\ \forall x.\bigvee\{x=a \mid a \in |A|\})$ where $|A|$ is the carrier of A, and $\mathscr{D}(A)$ is the first-order diagram of the expansion of A to a $\Sigma(|A|)$-algebra with the natural interpretation of the new constants. If $B \cong A$ then obviously $B \models \zeta(A)$.
Conversely, assume that B satisfies $\zeta(A)$. Thus, there is a valuation $v:|A| \rightarrow |B|$ such that $B \models_v \bigwedge \mathscr{D}(A)\ \&\ \forall x.\bigvee\{x=a \mid a \in |A|\}$ ($B \models_v \varphi$ means B satisfies φ under the valuation v; we are going to use this notation throughout the paper.) It is easy to see that v is an isomorphism between A and B. □

Note from the construction that in order to handle algebras of cardinality α it is enough to consider formulae with quantifiers binding α variables. It seems to be possible to sharpen this result (requiring only quantifiers binding less than α variables) using some kind of "back-and-forth" construction as in the proof of Scott's theorem in [Bar 73].

In practice it is desirable to avoid use of infinitary logic (although [MSV 83] argue for an approach to specification in which infinitary logic is central). What we are trying to do in the following is to obtain a balance between the power of the logic in use (the simpler the logic, the better) and the simplicity of the definition of observational equivalence.

It is obvious that using ground equations as observations we are not able to talk about junk at all. If we use equations with universally quantified variables, although we are able to say something about junk we cannot always distinguish between algebras which are intuitively not equivalent. For example, the two algebras

A: $$\begin{tikzcd} \bullet \arrow[r, "f"] & \bullet \arrow[r, "f"] & \cdots \\ \bullet \arrow[r, "f"'] & \bullet \arrow[r, "f"'] & \cdots \end{tikzcd}$$

B: $$\begin{tikzcd} \bullet \arrow[r, "f"] & \bullet \arrow[r, "f"] & \cdots \\ \bullet \arrow[loop right, "f"] \end{tikzcd}$$

cannot be distinguished by any equation with universally quantified variables (neither of them satisfy $\forall x.f(x)=x$) although if we go to first-order logic then the formula $\exists x.f(x)=x$ distinguishes between them. But even in the framework of first-order logic using only closed sentences, we are not able to deal with junk in a satisfactory way. We cannot even express such a basic property as its existence. For example, it is well-known that the standard model of arithmetic (the natural numbers) and non-standard models (the natural numbers with junk) satisfy exactly the same set of first-order sentences. Thus, there is no set of ground first-order observations which can distinguish between standard and non-standard models of arithmetic. To distinguish between these models using ground observations we need $L_{\omega_1\omega}$.

We are going to extend the definitions of the previous section by allowing free variables in observations. The idea is that these provide a way of referring to otherwise unnameable values. For example, it should be intuitively clear (and will be formalised below) that the observation $f(x)=x$ with free variable x distinguishes between the two algebras A and B above, and the set of observations $\{x=succ^n(0) \mid n\geq 0\}$ with free variable x distinguishes between standard and non-standard models of arithmetic. As in logic, we need a valuation of the free variables into the algebra under consideration to provide these names with interpretations.

Given a signature Σ, a set X of variables (of sorts in Σ), a set $\Phi(X)$ of Σ-formulae with free variables in X, and two Σ-algebras A,B there are a number of possible ways to define $A\equiv_{\Phi(X)}B$. For example, in [SW 83] and [ST 84] $A\equiv_{\Phi(X)}B$ was defined as follows:

$A\equiv_{\Phi(X)}B$ if there exist surjective valuations $v_A:X\to|A|$ and $v_B:X\to|B|$ such that for all $\varphi\in\Phi(X)$, $A\models_{v_A}\varphi$ iff $B\models_{v_B}\varphi$.

The justification for this definition is that v_A and v_B identify "matching parts" of A and B; each part of A must match some part of B and vice versa. But there are some problems with this definition. Technically, this relation is restricted to comparing algebras of cardinality less than or equal to that of X because of the surjectivity requirement on v_A and v_B. Also, we have to exclude algebras with empty carriers, (at least) on sorts in which X is non-empty; otherwise the valuations v_A and/or v_B cannot exist. Finally, in the "general" case in which models and the logic are arbitrary (see [ST 84]) this definition is rather messy and inelegant because of the difficulty of formulating in abstract terms the requirement of surjectivity.

We are going to concentrate on a different definition of observational equivalence. We define the observational equivalence relation in terms of a preorder.

Definition: For any signature Σ, set X of variables of sorts in Σ, set $\Phi(X)$ of Σ-formulae with free variables in X, and Σ-algebras A,B, A is *observationally reducible to* B *wrt* $\Phi(X)$, written $A\leq_{\Phi(X)}B$, if for any valuation $v_A:X\to|A|$ there exists a valuation $v_B:X\to|B|$ such that for all $\varphi\in\Phi(X)$, $A\models_{v_A}\varphi$ iff $B\models_{v_B}\varphi$.

Fact 16: For any signature Σ, set X of variables of sorts in Σ, and set $\Phi(X)$ of Σ-formulae with free variables in X, $\leq_{\Phi(X)}$ is a preorder on the class of Σ-algebras. □

Definition: For any signature Σ, set X of variables of sorts in Σ, set $\Phi(X)$ of Σ-formulae with free variables in X, and Σ-algebras A,B, A and B are *observationally equivalent wrt* $\Phi(X)$, written $A\equiv_{\Phi(X)}B$, if $A\leq_{\Phi(X)}B$ and $B\leq_{\Phi(X)}A$.

Although we are not going to restate all of them formally here again, facts 2-6 of section

2 hold for the preorder $\leq_{\Phi(X)}$ and facts 1-6 hold for the equivalence $\equiv_{\Phi(X)}$. For example, fact 3 may be reformulated for $\leq_{\Phi(X)}$ here as follows:

Fact 3': For any signature Σ, family of mutually disjoint sets $\{X_i\}_{i\in I}$ of variables of sorts in Σ, family $\{\Phi_i\}_{i\in I}$ of sets of Σ-formulae such that for $i\in I$, Φ_i has free variables in X_i, and Σ-algebras A,B, $A\leq_{\Phi_i(X_i)}B$ for all $i\in I$ implies $A\leq_{\Phi(X)}B$ where $\Phi = \bigcup_{i\in I}\Phi_i$ and $X = \bigcup_{i\in I}X_i$. □

However, because of the problems which empty carriers may cause, we have to be careful with the opposite direction of this implication, that is when discharging variables. Fact 2 should be reformulated as follows:

Fact 2': For any signature Σ, set X of variables of sorts in Σ, sets $\Phi(X)$ and $\Phi'(X)$ of Σ-formulae with free variables in X, and Σ-algebras A,B,
$\Phi(X)\supseteq\Phi'(X)$ and $A\leq_{\Phi(X)}B$ implies $A\leq_{\Phi'(X)}B$. □

Note that Φ and Φ' must formally have the same set X of free variables, even if the formulae in the smaller set Φ' do not use all of them. We can discharge such unnecessary variables only if X contains other variables of the same sorts, or if the algebras we are dealing with are guaranteed to have non-empty carriers of these sorts.

As in the previous section, we can define a specification-building operation **abstract** in terms of observational equivalence with exactly the same semantics:

Definition: For any specification SP, set X of variables of sorts in Sig[SP] and set $\Phi(X)$ of Sig[SP]-formulae with free variables in X

Sig[**abstract** SP **wrt** $\Phi(X)$] = Sig[SP]
Mod[**abstract** SP **wrt** $\Phi(X)$] = $\{ A \mid A\equiv_{\Phi(X)}B$ for some $B\in$Mod[SP] $\}$

Facts 7-12 still hold under this more general definition.

Note that we can give a sharper formulation of the facts which involve forming the closure $Cl(\Phi)$ of a set of formulae Φ. In the presence of free variables, besides conjunctions and negations it is tempting to allow the introduction of quantifiers here. We can redefine $Cl(\Phi(X))$ to be the closure of $\Phi(X)$ under negation, conjunction (possibly infinite), equivalence and uniform quantification, that is, $\varphi\in Cl(\Phi(X))$ implies $\forall X.\varphi\in Cl(\Phi(X))$ and $\exists X.\varphi\in Cl(\Phi(X))$. To prove that all facts are still true with this new definition of $Cl(\Phi(X))$, we have to show the following:

Fact 17: $\equiv_{Cl(\Phi(X))} \supseteq \equiv_{\Phi(X)}$ □

Note that only uniform quantification is allowed above. The above fact does not hold if we allow quantification over a proper subset of the set of free variables. For example, suppose Σ = **sorts** rat,bool **opns** <:rat,rat→bool. Let A and B be Σ-algebras corresponding to, respectively, open and closed intervals of rational numbers. Now consider $\Phi(X) = \{x<y \mid x,y\in X\}$. Obviously $A\equiv_{\Phi(X)}B$ but $A\models\forall x.\exists y.x<y)$ while $B\not\models\forall x.\exists y.x<y)$. Note also that even under this definition of closure, fact 13 does not hold for non-ground observations.

4 Proofs in structured specifications

An important issue connected with specifications is theorem proving. We would like to be able to prove theorems about a specification, that is, that certain sentences of the underlying logic hold in every model of a specification. As suggested by Guttag and Horning [GH 80] by proving that selected theorems hold we can understand specifications and gain confidence that they express what we want. Moreover, in order to do any kind of formal program development or verification (or even specification building, if parameterised specifications with requirements can be used) a theorem-proving capability is necessary.

In the context of structured specifications, we have to cope with two separate problems. First is how to prove theorems in theories of the underlying logic. Note that this task may be eased by the fact that our theories have structure, as this allows us to naturally disregard information which is probably irrelevant to what we are trying to prove. The other problem is dealing with the structure itself. What we need are inference rules for every specification-building operation which allow us to derive theorems about a combined specification from theorems about the components from which it was built. Note that the latter problem is not automatically reducible to the former because not all specifications are equivalent to (have the same class of models as) theories of the underlying logic [ST 84], let alone theories with finite presentations as required for use by a theorem prover.

For simple specification-building operations appropriate inference rules are given in [SB 83], for example

$$\text{thm in SP} \implies \text{thm in SP + SP'}$$

where "thm in SP" means $thm \in Th(Mod[SP])$, that is that the sentence *thm* holds in all models of the specification *SP*. The **abstract** specification-building operation defined in sections 2 and 3 is more difficult to handle. One problem is that in contrast to other specification-building operations it is not monotonic, in the sense that

$$\text{thm in SP} \not\Rightarrow \text{thm in } \textbf{abstract} \text{ SP } \textbf{wrt} \ldots$$

However, fact 12 and its analogue for observations with free variables (see section 3) says that the following inference rule is sound.

Inference rule: For any set $\Phi(X)$ of open formulae with variables in X,
$\text{thm in SP and } thm \in Cl(\Phi(X)) \implies \text{thm in } \textbf{abstract} \text{ SP } \textbf{wrt } \Phi(X)$

Moreover, for the case of ground observations (i.e. when X is the empty set), fact 13 shows that in some standard logics (first-order logic, infinitary logics) the above rule is in a sense complete when used together with inference rules for the underlying logic and the other specification-building operations. Note also that facts 7-12 provide us with some subsidiary inference rules; for example, fact 9 implies

$$\text{thm in } \textbf{abstract} \text{ SP } \textbf{wrt } \Phi \text{ and } \Phi \subseteq Cl(\Phi') \implies \text{thm in } \textbf{abstract} \text{ SP } \textbf{wrt } \Phi'$$

5 Behavioural equivalence — an example

In sections 2 and 3 we defined a very general and powerful notion of observational equivalence. In this section we look at a very important special case and we consider an example of its use. Namely, we restrict observations to equations between terms from some specified set; this gives an equivalence corresponding to the one used in the ASL specification language [SW 83]. A proper choice of the set of terms gives *behavioural equivalence* as informally discussed in the introduction.

Suppose that Σ is a signature and IN and OUT are subsets of the sorts of Σ. Now, consider all computations which take input from sorts IN and give output in sorts OUT; this set of computations corresponds to the set of Σ-terms of sorts OUT with variables of sorts IN. Consider the set $EQ_{OUT}(X_{IN})$ of equations between terms of the same sort in OUT having variables X_{IN} of sorts in IN. Two algebras are observationally equivalent with respect to $EQ_{OUT}(X_{IN})$ if they are behaviourally equivalent, that is they have matching input/output relations. Note that 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 83], [SW 83] and [GM 83] IN=OUT=OBS; and in [GGM 76] and [Kam 83] IN=ϕ and OUT=OBS. To denote the corresponding special case of **abstract** we use the following notation:

behaviour SP **with in** IN **out** OUT $=_{\text{def}}$ **abstract** SP **wrt** $EQ_{OUT}(X_{IN})$

This corresponds to *behavioural abstraction* as defined in ASL [SW 83].

As an example we are going to consider a simple language of expressions for arithmetical computation over the integers. This may be imagined as a small piece of a real programming language. We believe that the approach used below may be applied to other programming language constructs as well, leading towards the possible formal development of a compiler.

We assume that we are given some standard specifications of identifiers (Ident) with a sort *ident* and of the integers (Int) with the usual arithmetic operations. The (abstract) syntax of expressions is given by the following specification (we use the notation of the Clear specification language [BG 80][3]):

```
Expr = enrich Int + Ident by data sorts expr
                                  opns const : int → expr
                                       var : ident → expr
                                       plus, times : expr, expr → expr
                                       cond : expr, expr, expr → expr
```

The use of **data** above means that any model of Expr is a free extension of a model of Int + Ident. That is, the sort *expr* contains expressions built up using the newly-introduced operations. We could achieve the same effect using a *hierarchy constraint* [Bau 81] (cf. [SW 82] and [EWT 83]) together with the appropriate inequations.

To describe the semantics of expressions we need the additional concept of an environment from which the values of variables may be retrieved. This is described by the following (loose) specification:

```
Env = enrich Int + Ident by sorts env
                            opns lookup : env, ident → int
```

For the purpose of our example, no more than the existence of an operation *lookup* is required.

```
Eval = enrich Expr + Env by
         opns   eval : expr, env → int
         axioms ∀n:int, ρ:env. eval(const(n),ρ) = n
                ∀x:ident, ρ:env. eval(var(x),ρ) = lookup(ρ,x)
                ∀e,e':expr, ρ:env. eval(plus(e,e'),ρ) = eval(e,ρ) + eval(e',ρ)
                ∀e,e':expr, ρ:env. eval(times(e,e'),ρ) = eval(e,ρ) × eval(e',ρ)
                ∀e,e',e'':expr, ρ:env. eval(cond(e,e',e''),ρ) = eval(e'',ρ)  if eval(e,ρ) = 0
                                                              = eval(e',ρ)  otherwise
```

(We use an obvious notation to simplify the syntax of conditional axioms.)

The models of Eval are just the models of Expr with the expected semantics provided by the operation *eval*. The *cond* construct has the semantics of **if** _ **then** _ **else** _, where 0 (as the value of the first argument) is interpreted as false and any other value is interpreted as true. Note that the models of Eval are pretty well determined; in fact they are determined up to isomorphism given models of Ident and Env. Now imagine that we want to build a compiler which performs some source-level optimisation; for example, recognising that *times*(*const*(0),*e*) is just *const*(0). Such optimisations are not permitted by the

[3]But for the semantics of **derive**, see [SW 83].

specification above.

Two solutions to this dilemma are offered in the literature. First, [Wand 79] and [Kam 83] advocate the use of *final* models; if we adopt this approach (modifying the above specification appropriately) then every (final) model of Eval would satisfy $e = e'$ iff it satisfies $\forall \rho{:}env.\ eval(e,\rho) = eval(e',\rho)$, for all expressions e and e'. But this disallows non-optimal implementations since it requires that all possible optimisations are performed. Much worse, the specified models are actually not attainable since the optimisation required is not computable (this follows from a result in [Chu 36]).

Second, as advocated in e.g. [Ehr 79] and [EKMP 82] the notion of *implementation* of one specification by another should take care of this problem. Algebras with some optimisations are not models of the specification above but models of a specification which implements it. Unfortunately, the formal notions of implementation which have been suggested are rather complicated, and especially so in the context of loose and parameterised specifications. (Note that the specification above may be viewed as parameterised by Ident.)

We adopt neither of these solutions. Instead, we argue that the specification Eval as given above is not really what we intend. When we specify a program what we are really interested in is its behaviour, that is the answers which we obtain when the program is applied to the various possible inputs. The specification Eval says more than that; it dictates the structure of internal data. We can obtain the class of models having the behaviour which Eval specifies (rather concretely) by applying the **behaviour** operation for the appropriate choice of input and output sorts:

Eval-we-really-want = **behaviour** Eval **with in** {int,ident,env} **out** {int}

The inference rule for **abstract** given in section 4 may be applied here to show e.g. that

$\forall$e,e':expr, ρ:env. eval(plus(e,e'),ρ) = eval(plus(e',e),ρ)

is a theorem in Eval-we-really-want, since it is a theorem of Eval and is in the closure of the set of observations we are using here.[4]

The ability to specify classes of algebras up to behavioural equivalence (as in Eval-we-really-want) allows us to greatly simplify our formal view of what an implementation is. Proceeding from a specification to a program means making a series of design decisions, each of which amounts to a restriction on the class of models. Such design decisions are choice of data structures, choice of algorithms, and choice between alternatives which the specification leaves open.

Thus, a simple but natural notion of implementation is as follows.

Definition: A specification SP is *implemented by* a specification SP', written SP$\rightsquigarrow$SP', if Mod[SP'] $\subseteq$ Mod[SP].

It is easy to see that the above implementation relation is transitive (SP$\rightsquigarrow$SP' and SP'$\rightsquigarrow$SP" implies SP$\rightsquigarrow$SP"), i.e. that it can be composed *vertically* (see [GB 80]). This means that a specification can be refined gradually. Furthermore, this implementation relation can be composed *horizontally* [GB 80] as well[5] [SW 83] (SP1$\rightsquigarrow$SP1' and SP2$\rightsquigarrow$SP2' implies SP1+SP2$\rightsquigarrow$SP1'+SP2' and similarly for the other specification-building operations). This means that specifications can be refined in a modular fashion. This is in contrast to the more complicated notions of implementation mentioned earlier for which these

[4] For technical reasons (see [GM 81]) we assume that there are constants of sort *ident*.

[5] provided that all specification-building operations are monotonic (with respect to model classes), which is the case for the specification-building operations defined in e.g. Clear [BG 80], LOOK [ETLZ 82], ASL [SW 83], and for **abstract** and **behaviour** as defined above.

properties do not hold in general.

The following specification is an implementation of Eval-we-really-want:

```
Eval' =
  let Ev0 = enrich Eval by
             opns   optplus, opttimes : expr, expr → expr
                    optcond : expr, expr, expr → expr
             axioms ∀e,e':expr. optplus(e,e')
                              = e'                     if e = const(0)
                              = e                      if e' = const(0)
                              = opttimes(const(2),e)   if e = e'
                              = plus(e,e')             otherwise
                    ∀e,e':expr. opttimes(e,e')
                              = const(0)               if e = const(0) or e' = const(0)
                              = e'                     if e = const(1)
                              = e                      if e' = const(1)
                              = times(e,e')            otherwise
                    ∀e,e',e":expr. optcond(e,e',e")
                              = e'                     if e = const(n) and n ≠ 0
                              = e"                     if e = const(0)
                              = e'                     if e' = e"
                              = cond(e,e',e")          otherwise
  in derive signature Eval
       from Ev0
       by const is const
          var   is var
          plus  is optplus
          times is opttimes
          cond  is optcond
          eval  is eval
```

Eval' specifies the syntax and semantics of our expression language, requiring that certain source-level optimisations (constant folding) be carried out.

In order to prove that Eval' implements Eval-we-really-want we have to show:

Claim: Mod[Eval-we-really-want] $\supseteq$ Mod[Eval']

To prove this we have to show that any model of Eval' is behaviourally equivalent to a model of Eval (with respect to input sorts {int,ident,env} and output sort {int}). This boils down to showing that the value of an expression (as given by *eval*) is the same as the value of its optimisation in any environment (see the long version of this paper for details). □

A different way of proving that two algebras are behaviourally equivalent is suggested in [Sch 83]; in this approach, a relation (called a *correspondence*) between the corresponding carriers is set up explicitly and proved to satisfy a kind of homomorphism property.

6 Concluding remarks

In the previous sections we have been rather vague about what we mean by a "formula". We have mentioned formulae of equational logic, first-order logic and infinitary logic. Moreover, although we have been using the standard notion of many-sorted algebra as in [ADJ 76], this was mostly in order to take advantage of the reader's intuition; in fact, we made use of very few formal properties of algebras. This means that in place of the standard notion we could have used for example partial or continuous algebras. We could even change both the notions of signature and of algebra to deal with errors or coercions.

The notion of an *institution* [GB 83] provides a tool for dealing with any of these different notions of a logical system for writing specifications. An institution comprises definitions of signature, model (algebra), sentence and a satisfaction relation satisfying a few minimal consistency conditions. (For a similar but more logic-oriented approach see [Bar 74].) By basing our definitions (of observational equivalence etc.) on an arbitrary institution we can avoid choosing particular definitions of these underlying notions and do everything at an adequately general level. It is possible to define the semantics of a specification language in an arbitrary institution; see [BG 80] and [ST 84].

We encounter no problems at all in generalising the contents of section 2 (on ground observations) to an arbitrary institution. Moreover, facts 1-12 still hold. (Fact 13 holds for institutions with some simple closure properties. Fact 14 may be generalised if we equip institutions with some notion of reachability along the lines of [Tar 84].)

In order to deal with the general case of observations containing free variables we have first of all to provide a notion of an open formula and a valuation of free variables in the framework of an arbitrary institution. Although sentences as they are used in the definition of an institution above are always closed, this may be done (see [ST 84]). Then the contents of section 3 may be generalised as well; see the longer version of this paper for details.

By exploring the properties of a primitive but powerful and general notion such as observational equivalence and then deriving the more directly useful concept of behavioural equivalence as a special case, we are following in the footsteps of earlier work on kernel specification-building operations [Wir 82,83], [SW 83], [ST 84]. Our ultimate interest is not in the primitive notions themselves but rather in the useful higher-level constructs which can be expressed in their terms. By carefully investigating the primitives we hope to gain insights which can be applied to the derived constructs.

The material in this paper could provide the basis for high-level specification languages such as one in which every specification is surrounded by an implicit (and invisible) application of **behaviour** with respect to input and output sorts appropriate to the context. Such a language is presented in [ST 85]. An issue we have not discussed is the connection between behavioural equivalence/abstraction and parameterisation of specifications. A different approach to the problem of specifying software modules which integrates parameterisation and implementation is given in [Ehrig 84]. We have not yet investigated thoroughly the interaction between **behaviour** and other specification-building operations, although a start in this direction is given by facts 5 and 6.

Acknowledgements

Our thanks to Rod Burstall for many instructive discussions and encouragement and to Martin Wirsing for the collaboration which started us on this line of work and for helpful comments on an earlier version. Support was provided by the Science and Engineering Research Council.

7 References

[ADJ 76] Goguen, J.A., Thatcher, J.W. and Wagner, E.G. An initial algebra approach to the specification, correctness, and implementation of abstract data types. IBM research report RC 6487. Also in: Current Trends in Programming Methodology, Vol. 4: Data Structuring (R.T. Yeh, ed.), Prentice-Hall, pp. 80-149 (1978).

[Bar 73] Barwise, J. Back and forth through infinitary logic. In: Studies in Mathematics, Vol. 8: Studies in Model Theory (M.D. Morley, ed.), Mathematical Assoc. of America, pp. 5-34.

[Bar 74] Barwise, J. Axioms for abstract model theory. Annals of Math. Logic 7, pp. 221-265.

[Bau 81] Bauer, F.L. *et al* (the CIP Language Group) Report on a wide spectrum language for program specification and development (tentative version). Report TUM-

I8104, Technische Univ. München.

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

[BG 82] Burstall, R.M. and Goguen, J.A. Algebras, theories and freeness: an introduction for computer scientists. Proc. 1981 Marktoberdorf NATO Summer School, Reidel.

[BMS 80] Burstall, R.M., MacQueen, D.B. and Sannella, D.T. HOPE: an experimental applicative language. Proc. 1980 LISP Conference, Stanford, California, pp. 136-143.

[Chu 36] Church, A. An unsolvable problem of elementary number theory. American Journal of Mathematics 58, pp. 345-363.

[Ehr 79] Ehrich, H.-D. On the theory of specification, implementation, and parametrization of abstract data types. Report 82, Abteilung Informatik, Univ. of Dortmund. Also in: JACM 29, 1, pp. 206-227 (1982).

[Ehrig 84] Ehrig, H. An algebraic specification concept for modules (draft version). Report 84-04, Institut für Software und Theoretische Informatik, Technische Univ. Berlin.

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

[ETLZ 82] Ehrig, H., Thatcher, J.W., Lucas, P. and Zilles, S.N. 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. Algebraic specifications with generating constraints. Proc. 10th ICALP, Barcelona. Springer LNCS 154, pp. 188-202.

[End 72] Enderton, H.B. A Mathematical Introduction to Logic. Academic Press.

[GGM 76] Giarratana, V., Gimona, F. and Montanari, U. Observability concepts in abstract data type specification. Proc. 5th MFCS, Gdansk. Springer LNCS 45.

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

[GB 83] Goguen, J.A. and Burstall, R.M. Introducing institutions. Proc. Logics of Programming Workshop, Carnegie-Mellon. Springer LNCS 164, pp. 221-256.

[GM 81] Goguen, J.A. and Meseguer, J. Completeness of many-sorted equational logic. SIGPLAN Notices 16(7), pp. 24-32; extended version to appear in Houston Journal of Mathematics.

[GM 82] Goguen, J.A. and Meseguer, J. Universal realization, persistent interconnection and implementation of abstract modules. Proc. 9th ICALP, Aarhus, Denmark. Springer LNCS 140, pp. 265-281.

[GM 83] Goguen, J.A. and Meseguer, J. An initiality primer. Draft report, SRI International.

[GH 80] Guttag, J.V. and Horning, J.J. Formal specification as a design tool. Proc. ACM Symposium on Principles of Programming Languages, Las Vegas, pp. 251-261.

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

[Karp 64] Karp, C.R. Languages with Expressions of Infinite Length. North-Holland.

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

[MSV 83] Maibaum, T.S.E., Sadler, M.R. and Veloso, P.A.S. Logical implementation. Technical report, Department of Computing, Imperial College.

[Pep 83] Pepper, P. On the correctness of type transformations. Talk at 2nd Workshop on Theory and Applications of Abstract Data Types, Passau.

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

[SB 83] Sannella, D.T. and Burstall, R.M. Structured theories in LCF. Proc. 8th Colloq. on Trees in Algebra and Programming, L'Aquila, Italy. Springer LNCS 159, pp. 377-391.

[ST 84] Sannella, D.T. and Tarlecki, A. Building specifications in an arbitrary institution. Proc. Intl. Symposium on Semantics of Data Types, Sophia-Antipolis. Springer LNCS 173, pp. 337-356.

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

Orleans.

[SW 82] Sannella, D.T. and Wirsing, M. Implementation of parameterised specifications. Report CSR-103-82, Dept. of Computer Science, Univ. of Edinburgh; extended abstract in: Proc. 9th ICALP, Aarhus, Denmark. Springer LNCS 140, pp. 473-488.

[SW 83] Sannella, D.T. and Wirsing, M. A kernel language for algebraic specification and implementation. Report CSR-131-83, Dept. of Computer Science, Univ. of Edinburgh; extended abstract in: Proc. Intl. Conf. on Foundations of Computation Theory, Borgholm, Sweden. Springer LNCS 158, pp. 413-427.

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

[Scott 65] Scott, D. Logic with denumerably long formulas and finite strings of quantifiers. In: Theory of Models. North-Holland, pp. 329-341.

[Tar 84] Tarlecki, A. Free constructions in abstract algebraic institutions. Draft report, Dept. of Computer Science, Univ. of Edinburgh.

[Wand 79] Wand, M. Final algebra semantics and data type extensions. JCSS 19, pp. 27-44.

[Wir 82] Wirsing, M. Structured algebraic specifications. Proc. AFCET Symp. on Mathematics for Computer Science, Paris, pp. 93-107.

[Wir 83] Wirsing, M. Structured algebraic specifications: a kernel language. Habilitation thesis, Technische Univ. München.