Observability concepts in abstract data type specification, 30 years later*

Donald Sannella[1] and Andrzej Tarlecki[2,3]

[1] Laboratory for Foundations of Computer Science, University of Edinburgh
[2] Institute of Informatics, Warsaw University
[3] Institute of Computer Science, Polish Academy of Sciences

Abstract. We recall the contribution of Montanari's paper [GGM76] and sketch a framework for observable behaviour specification that blends some of these early ideas, seen from a more modern perspective, with our own approach.

1 Introduction

The starting point for this work is a brief paper [GGM76] coauthored by Ugo Montanari and published in 1976. This appears to be the first of many papers to study observational aspects of the algebraic approach to software specification and development, where the overall idea is that one should regard a specification of a system as constraining its observable behaviour, and nothing more. Such a view is required to cope with many examples. However, it adds significant technical complexities to the simple and elegant algebraic approach. Some of these remain unresolved today, even after 30 years of research.

[GGM76] starts by challenging the initial algebra approach to specifications of abstract data types, then recently introduced by early versions of [GTW78]. Most importantly, [GGM76] points out that not all sorts of data in a data type play the same role: one should separate the given, "old" sorts from the "new" ones, to be specified and implemented. What really matters then is the behaviour of the data type as viewed via these old sorts only; the implementation details of the new sorts play a secondary role. Such *observable behaviour* is captured by the evaluation function restricted to terms that are of old sorts, but in general use the new operations and involve new sorts internally. Another crucial insight in [GGM76] is that in general there are many non-isomorphic algebras that display the same observable behaviour. They show that the set of isomorphism classes of such algebras (limited to the ones generated by the old sorts) forms a complete lattice — a nice technical result which, however, is not used to insist that any such specific algebra is always chosen (as in the initial [GTW78] or final [Wan79] algebra approaches) since all of them are equally adequate implementations of the given observable behaviour. Such behaviours are specified in [GGM76] by

* This work has been partially supported by European projects IST-2005-015905 MOBIUS (DS, AT) and IST-2005-016004 SENSORIA (AT).

giving a partial evaluation function, which assigns values to some terms of old sorts only, marking the others as "don't care" cases (indicated by assigning to them a special "value" α, a notation that we will maintain here). The latter captures the situation where the specifier permits the behaviour to be chosen arbitrarily (but consistently with other choices) in any particular implementation. Particular implementations for such a behaviour specification in [GGM76] are captured as (generated) algebras that conform to the specification in the obvious sense.

Quite a few points made in [GGM76] were very insightful in their historical context. This is the first place we know of where several key ideas appear, including some that underlie most of our own contributions to the area. First, the stress on the need for loose specifications, which need not determine behaviour unambiguously (up to isomorphism) was of key importance. The results on the lattice properties of the class of models for a given observable behaviour initiated a line of research in this direction, including a debate on the issue of initial vs. final interpretation of algebraic specifications. One aspect which disappeared in later work was the method of presenting specifications by using an explicitly given set of data on which the data type is based, with behaviour specified by indicating the results of evaluation of some terms, while explicitly marking others as "don't care" cases. The authors' techniques turn out to be very close to "abstract model specifications" in the style of VDM [Jon80]. The main contribution though is the idea of limiting specifications to observable parts of behaviour only, thus introducing observability aspects to algebraic specification.

The pioneering role of [GGM76] is underlined by the fact that it cites just 12 references, some of them unpublished, with only a few concerning algebraic specifications. Hardly any other papers in the field could have been mentioned then: at the time, this is essentially all that there was! This has to be contrasted with the outburst of work in the area in the following years, as for instance summarised in the bibliography [BKL+91] some 15 years later, or in the overview presentations of the field in [Wir90] or the more recent [AKKB99]. One important line of activity concerned observability aspects, with an extensive literature of its own, including [Rei81] and numerous papers presenting further developments in various directions, at diverse levels of abstraction. This includes for instance the popular *hidden algebra* framework [GM00] and our own work [ST87] aimed at bringing this closer to logical characterisation via elementary equivalence, with [BH06] offering a recent elegant approach benefiting from all this experience.

We reiterate some of the ideas presented in [GGM76] here, looking back at more than 30 years of work on algebraic specification, and trying to blend what happened with these ideas with our current personal perspective. We sketch a framework for observable behaviour specification and development, reconsidering some of the work presented earlier [ST88b,BST02,BST08] in a different technical setting. It is reassuring that, after shifting to quite a different specification technology, inspired by [GGM76], our basic ideas on system specification, architectural design and development under an observational view of specifications still stand.

2 Algebraic preliminaries

Signatures and signature morphisms are as usual, except for the treatment of the distinguished sort *bool*.

Definition 2.1. *A* signature $\Sigma = \langle S, \Omega \rangle$ *consists of a set S of* sort names *and an $S^* \times S$-indexed set Ω of* operation names, *where $f \in \Omega_{\langle s_1 \cdots s_n, s\rangle}$ is written $f : s_1 \times \cdots \times s_n \to s$. We require that bool $\in S$ and that no operations in Ω take arguments in bool. If A is an S-sorted set, then $\Sigma(A)$ denotes the signature obtained from Σ by adding the elements of A to Ω as constants.*

If $\Sigma' = \langle S', \Omega' \rangle$, then a signature morphism $\sigma : \Sigma \to \Sigma'$ *consists of a mapping of sort names, $\sigma : S \to S'$, and an $S^* \times S$-sorted mapping of operation names, with $\sigma_{\langle s_1 \cdots s_n, s\rangle} : \Omega_{\langle s_1 \cdots s_n, s\rangle} \to \Omega'_{\langle \sigma(s_1) \cdots \sigma(s_n), \sigma(s)\rangle}$, such that $\sigma(s) = bool$ iff s is bool.*

We regard *bool* as the sort of logical meta-values, where operations that deliver results in *bool* are like predicates. Forbidding operations taking arguments in *bool* corresponds to the fact that applying a predicate to a tuple of terms would normally yield an atomic formula, not a term. We treat predicates here as operations, with this restriction, rather than as relations, for the sake of technical convenience. Observations (see Sect. 4) will be terms of sort *bool*.

Algebras and their homomorphisms are defined as usual, except that we fix the interpretation of the distinguished sort *bool* to be the set $\mathbb{B} = \{true, false\}$.

Definition 2.2. *Given a signature $\Sigma = \langle S, \Omega \rangle$, a Σ-*algebra $\mathbf{A}$ *consists of an S-sorted carrier set A and, for each operation name $f : s_1 \times \cdots \times s_n \to s$, a function $f_{\mathbf{A}} : A_{s_1} \times \cdots \times A_{s_n} \to A_s$. We require that $A_{bool} = \mathbb{B}$.*

*A Σ-*homomorphism $m : \mathbf{A} \to \mathbf{B}$ *between Σ-algebras $\mathbf{A}$ and $\mathbf{B}$ is an S-sorted family of functions $m_s : A_s \to B_s$, $s \in S$, that preserve the values of operations, as usual. We require that m_{bool} is the identity on $\mathbb{B}$.*

*Given a signature morphism $\sigma : \Sigma \to \Sigma'$, for any Σ'-algebra $\mathbf{A}'$, its σ-*reduct *is the Σ-algebra $\mathbf{A} = \mathbf{A}'|_\sigma$ given by $A_s = A'_{\sigma(s)}$ for $s \in S$, and $f_{\mathbf{A}} = \sigma(f)_{\mathbf{A}'}$ for $f \in \Omega$. Reducts of Σ'-homomorphisms and of S'-sorted sets as well as of (S'-sorted) functions and relations between them are defined analogously.*

Signatures and their morphisms form a category, which is cocomplete. Σ-algebras and homomorphisms between them form a category (which is also cocomplete, with the algebra $\mathbf{T}_{\boldsymbol{\Sigma}}$ of ground Σ-terms as the initial object). For any signature morphism $\sigma : \Sigma \to \Sigma'$, σ-reduct is a functor. Moreover, the assignments of the categories of algebras to signatures, and of reduct functors to signature morphisms form a (contravariant) functor from the category of signatures to the category of "all" categories. This functor is continuous, so that in particular the following amalgamation lemma holds:

Lemma 2.3. *Consider a pushout in the category of signatures.*

$$\begin{tikzcd} \Sigma_1 \arrow[r, "\sigma'_2"] & \Sigma' \\ \Sigma \arrow[u, "\sigma_1"] \arrow[r, "\sigma_2"'] & \Sigma_2 \arrow[u, "\sigma'_1"'] \end{tikzcd}$$

Then for any Σ_1-algebra $\mathbf{A_1}$ and Σ_2-algebra $\mathbf{A_2}$ with common Σ-reduct $\mathbf{A_1}|_{\sigma_1} = \mathbf{A_2}|_{\sigma_2}$, there exists a unique Σ'-algebra $\mathbf{A}'$ such that $\mathbf{A}'|_{\sigma'_2} = \mathbf{A_1}$ and $\mathbf{A}'|_{\sigma'_1} = \mathbf{A_2}$; and similarly for homomorphisms.

Given a signature $\Sigma = \langle S, \Omega \rangle$ and an S-sorted set A, we will consider the set $T_\Sigma(A)$ of Σ-terms with "variables" in A (when A is empty, we write T_Σ for $T_\Sigma(A)$). Equivalently we could take $T_{\Sigma(A)}$, where elements of A are considered as additional constants. The distinction will be disregarded whenever convenient. Terms that are in A will be referred to as *data*; the others as *non-data*.

Given a signature morphism $\sigma\colon \Sigma \to \Sigma'$, S-sorted set A and S'-sorted set A' such that $A \subseteq A'|_\sigma$, σ induces the translation $\sigma : T_\Sigma(A) \to T_{\Sigma'}(A')$ in the obvious way: $\sigma(a) = a$ for data terms $a \in A$, and extending this to non-data terms by replacing each Σ-operation name f with Σ'-operation name $\sigma(f)$.

By a $\Sigma(A)$-context we mean any Σ-term t that in addition to the operation names from Σ and data from A may contain an occurrence of a special variable $\Box$. Then, for any term $t' \in T_\Sigma(A)$, we write $t(t')$ for the term in $T_\Sigma(A)$ obtained by substituting t' for $\Box$ in t.[1]

3 Behaviours and behaviour specifications

Inspired by the notion of a (complete) specification in [GGM76] as a "black-box" view of models, we will not deal explicitly with algebras here, but rather concentrate on the study of their *behaviours*. Let $\Sigma = \langle S, \Omega \rangle$.

Definition 3.1. *A Σ-*behaviour *is an S-sorted carrier set A together with an S-sorted* evaluation function $ev\colon T_\Sigma(A) \to A$ *such that: $ev(a) = a$ for all data $a \in A$; if $ev(t') = a'$ then $ev(t(a')) = ev(t(t'))$ for all terms $t' \in T_\Sigma(A)$ and $\Sigma(A)$-contexts t; and $A_{bool} = \mathbb{B}$. We will use evaluation functions ev to refer to behaviours, with carriers left implicit.*

A knowledgeable reader will recognise the notion of an algebra for the monad T_Σ. In this definition, it suffices to consider contexts t of the form $f(a_1, \ldots, \Box, \ldots, a_n)$ for $a_1, \ldots, a_n \in A$. Examples for this and other definitions will come in Sect. 5.

Definition 3.2. *Given Σ-behaviours $ev_1\colon T_\Sigma(A_1) \to A_1$ and $ev_2\colon T_\Sigma(A_2) \to A_2$, a* homomorphism $m\colon ev_1 \to ev_2$ *is an S-sorted function $m\colon A_1 \to A_2$, such that m_{bool} is the identity function on $\mathbb{B}$, and if $m(ev_1(t_1)) = ev_2(t_2)$ then $m(ev_1(t(t_1))) = ev_2(\hat{t}(t_2))$ for all $\Sigma(A_1)$-contexts t, $\Sigma(A_1)$-terms t_1 and $\Sigma(A_2)$-terms t_2, where $\hat{t}$ results from t by replacing data $a \in A_1$ by $m(a) \in A_2$.*

In this definition, it is once again sufficient to consider contexts t of the form $f(a_1, \ldots, \Box, \ldots, a_n)$ for $a_1, \ldots, a_n \in A_1$, and t_1 and t_2 that are data terms.

As usual, semantics (behaviours) can be translated along signature morphisms in the opposite direction to the translation of syntax (terms):

Definition 3.3. *Consider a signature morphism $\sigma\colon \Sigma \to \Sigma'$ and Σ'-behaviour $ev'\colon T_{\Sigma'}(A') \to A'$. The σ-*reduct *of ev' is the Σ-behaviour $ev'|_\sigma\colon T_\Sigma(A) \to A$ where $A = A'|_\sigma$ and for $t \in T_\Sigma(A)$, $(ev'|_\sigma)(t) = ev'(\sigma(t))$.*

[1] To be precise, this requires a careful identification of the sort for the variable $\Box$ and the term t' — whenever convenient, we will continue omitting such details here.

Proposition 3.4. *Given a signature morphism $\sigma\colon \Sigma \to \Sigma'$, the σ-reduct $m|_\sigma$ of any homomorphism $m\colon ev'_1 \to ev'_2$ between Σ'-behaviours is a homomorphism between their σ-reducts, $m|_\sigma\colon ev'_1|_\sigma \to ev'_2|_\sigma$.*

There is a 1–1 correspondence between Σ-behaviours and Σ-algebras, and between homomorphisms as above and ordinary homomorphisms on algebras, as recalled in Sect. 2. This gives an isomorphism between the category of Σ-behaviours and the category of Σ-algebras, and carries over to the reduct functors determined by signature morphisms.

Σ-behaviours are specified by indicating what the values of certain terms should be, while explicitly indicating that the values of other terms are not constrained. We use α for the latter "don't care" case, following [GGM76].

Definition 3.5. *A* Σ-behaviour specification *is an S-sorted carrier set A together with an S-sorted function $h\colon T_\Sigma(A) \to A \cup \{\alpha\}$ such that: $h(a) = a$ for all data $a \in A$; if $h(t') = a'$ then $h(t(a')) = h(t(t'))$ for all terms $t' \in T_\Sigma(A)$ and $\Sigma(A)$-contexts t; and $A_{bool} = \mathbb{B}$. We will use functions h to refer to behaviour specifications, leaving their carriers implicit.*

It is important to understand that α ("don't care") does not mean that any choice of value will do; as we will see, the choice taken needs to respect the values of those terms that are specified as non-α.

There are two natural orderings on behaviour specifications, both reflecting the degree to which behaviour is constrained.

Definition 3.6. *Let $h_1\colon T_\Sigma(A_1) \to A_1 \cup \{\alpha\}$ and $h_2\colon T_\Sigma(A_2) \to A_2 \cup \{\alpha\}$ be Σ-behaviour specifications, and let $ev\colon T_\Sigma(A) \to A$ be a Σ-behaviour.*

1. *h_2* refines *h_1, written $h_1 \rightsquigarrow h_2$, if $A_1 \subseteq A_2$ and h_2 conforms to h_1, that is: for each $t \in T_\Sigma(A_1)$, $h_2(t) = h_1(t)$ whenever $h_1(t) \neq \alpha$. Then ev* satisfies *h_1 if ev (viewed as a behaviour specification) refines h_1. We write $Mod(h_1)$ for the class of all Σ-behaviours that satisfy h_1.*
2. *h_2* strongly refines *h_1 if h_2 refines h_1 and $A_1 = A_2$. Then $ev\colon T_\Sigma(A) \to A$* strongly satisfies *$h_1$ if ev strongly refines h_1, which requires $A_1 = A$.*

For any Σ-behaviour specification $h\colon T_\Sigma(A) \to A \cup \{\alpha\}$ we define its *free extension* by adding a new value for each of the "don't care" cases. It corresponds to the "initial symbolic representation" of h from [GGM76] (with all sorts viewed as "old") and is constructed as follows. Let I extend A by all non-data terms $t \in T_\Sigma(A)$ such that all subterms t' of t with $h(t') \neq \alpha$ are data. This not only requires that $h(t) = \alpha$, but also that all subterms of t are evaluated as far as determined by h. Then, the free extension of h is the only function ev_I : $T_\Sigma(I) \to I$ such that $ev_I(t) = h(t)$ for all terms $t \in T_\Sigma(A)$ with $h(t) \neq \alpha$, $ev_I(t) = t$ for all terms t in $I \setminus A$, and $ev_I(t(t')) = ev_I(t(ev_I(t')))$ for all contexts t and terms t'. Now, ev_I is not necessarily a Σ-behaviour, because I_{bool} need not be $\mathbb{B}$. However, we can obtain a Σ-behaviour from ev_I by choosing a function $f\colon I_{bool} \to \mathbb{B}$ that extends identity on $\mathbb{B}$. Let then $\hat{f}$ extend f to an S-sorted function that is the identity on all sorts other than *bool*, and let $\hat{I}$ be an S-sorted set such that $\hat{I}_s = I_s$ for all sorts $s \neq bool$ and $\hat{I}_{bool} = \mathbb{B}$. Then $ev^f_h \colon T_\Sigma(\hat{I}) \to \hat{I}$, such that $ev^f_h(t) = \hat{f}(ev_I(t))$ for $t \in T_\Sigma(\hat{I})$, is a Σ-behaviour that satisfies h. This proves the following:

Lemma 3.7. *Every Σ-behaviour specification $h \colon T_\Sigma(A) \to A \cup \{\alpha\}$ is satisfiable.*

It is not the case that any behaviour specification h is *strongly* satisfiable, unless we add the requirement that whenever $h(t_1) = \alpha$ then there exists some $a \in A$ such that if $h(t(t_1)) \neq \alpha$ then $h(t(a)) = h(t(t_1))$. (See condition (c) in the definition of specification in [GGM76].)

Lemma 3.8. *For Σ-behaviour specifications h_1 and h_2, $h_1 \rightsquigarrow h_2$ iff $Mod(h_2) \subseteq Mod(h_1)$.*

For technical convenience, let us introduce an additional special specification $\varnothing$ that is not satisfied by any behaviour, $Mod(\varnothing) = \varnothing$. We extend our definition of refinement to cover $\varnothing$ in the natural way, so that Lemma 3.8 still holds.

Complex behaviour specifications can be built in a structured way from simpler specifications, using standard specification-building operations such as derive (reduct), translate, union, defined in terms of their model classes in [ST88a]. For behaviour specifications of the form considered here, these kernel specification-building operations may be defined "internally".

Definition 3.9. *Reduct, translation and union of behaviour specifications are defined as follows:*

1. *Given a signature morphism $\sigma : \Sigma \to \Sigma'$ and Σ'-behaviour specification $h' : T_\Sigma(A') \to A' \cup \{\alpha\}$, its σ-*reduct *$h'|_\sigma$ is the Σ-behaviour specification $h : T_\Sigma(A|_\sigma) \to A|_\sigma \cup \{\alpha\}$ defined by $h(t) = h'(\sigma(t))$ for all $t \in T_\Sigma(A|_\sigma)$.*
2. *Given a signature morphism $\sigma : \Sigma \to \Sigma'$ and a Σ-behaviour specification $h : T_\Sigma(A) \to A \cup \{\alpha\}$, let $A'_{s'} = \bigcup\{A_s \mid \sigma(s) = s'\}$ for all sorts s' in Σ'. First, we define an auxiliary relation between terms $t' \in T_{\Sigma'}(A')$ and data $a' \in A'$, written h via σ forces t' to a', as the least relation such that*
   (a) *if $h(t) = a$ then h via σ forces $\sigma(t)$ to a, and*
   (b) *if h via σ forces t'' to a'' and h via σ forces $t'(a'')$ to a' then h via σ forces $t'(t'')$ to a'.*

   *Then the σ-*translation *$\sigma(h)$ of h is $\varnothing$ if for some term $t' \in T_{\Sigma'}(A')$ and two distinct data $a', a'' \in A'$, h via σ forces t' to a' and h via σ forces t' to a''. Otherwise, $\sigma(h)$ is the Σ'-behaviour specification $h' : T_\Sigma(A') \to A' \cup \{\alpha\}$ such that for each term $t' \in T_{\Sigma'}(A')$, $h'(t') = a'$ if h via σ forces t' to a' and $h'(t') = \alpha$ if such $a' \in A'$ does not exist.*
3. *Given Σ-behaviour specifications $h_1 : T_\Sigma(A_1) \to A_1 \cup \{\alpha\}$ and $h_2 : T_\Sigma(A_2) \to A_2 \cup \{\alpha\}$, first define an auxiliary relation between terms $t \in T_\Sigma(A_1 \cup A_2)$ and data $a \in (A_1 \cup A_2)$, written $\{h_1, h_2\}$ forces t to a, as the least relation such that*
   (a) *if $h_1(t) = a$ or $h_2(t) = a$ then $\{h_1, h_2\}$ forces t to a and*
   (b) *if $\{h_1, h_2\}$ forces t_0 to a_0 and $\{h_1, h_2\}$ forces $t(a_0)$ to a then $\{h_1, h_2\}$ forces $t(t_0)$ to a.*

Then, the union $h_1 + h_2$ *of* h_1 *and* h_2 *is* $\varnothing$ *if for some term* $t \in T_\Sigma(A_1 \cup A_2)$ *and two distinct data* $a, a' \in (A_1 \cup A_2)$, $\{h_1, h_2\}$ *forces* t *to* a *and* $\{h_1, h_2\}$ *forces* t *to* a'. *Otherwise,* $h_1 + h_2$ *is the* Σ*-behaviour specification* $h : T_\Sigma(A_1 \cup A_2) \to (A_1 \cup A_2) \cup \{\alpha\}$ *such that for each term* $t \in T_\Sigma(A_1 \cup A_2)$, $h(t) = a$ *if* $\{h_1, h_2\}$ *forces* t *to* a *and* $h(t) = \alpha$ *if such* $a \in (A_1 \cup A_2)$ *does not exist.*

Theorem 3.10. *1. Given a signature morphism* $\sigma : \Sigma \to \Sigma'$ *and* Σ'*-behaviour specification* h', $Mod(h'|_\sigma) \supseteq Mod(h')|_\sigma$, *where* $Mod(h')|_\sigma$ *is the class of all* σ*-reducts of* Σ'*-behaviours in* $Mod(h')$.

2. Given a signature morphism $\sigma : \Sigma \to \Sigma'$ *and* Σ*-behaviour specification* h, $Mod(\sigma(h)) = Mod(h)|_\sigma^{-1}$, *where* $Mod(h)|_\sigma^{-1}$ *is the class of all* Σ'*-behaviours with* σ*-reducts in* $Mod(h)$.

3. Given two Σ*-behaviour specifications* h_1 *and* h_2, *we have* $Mod(h_1 + h_2) = Mod(h_1) \cap Mod(h_2)$.

Note that for some (even injective) signature morphisms $\sigma : \Sigma \to \Sigma'$ and Σ'-behaviour specifications h' there may be no Σ-behaviour specification h such that $Mod(h) = Mod(h')|_\sigma$.

As usual, the above are just "kernel" operations on specifications, which underly more complex ones that are closer to what will be used in practical examples. For instance, if h is a Σ-behaviour specification, then we can write behaviour specifications over signatures extending Σ as follows (also permitting self-explanatory notational variants whenever convenient):

h **then** *signature-extension* **with** *sort-definitions* **and** *behaviour-definition*

Here, *signature-extension* is an extension of Σ, possibly contributing new sort names S' and operation names over $(S \cup S')^* \times (S \cup S')$, resulting in a new signature Σ', *sort-definitions* provides carrier definitions for the sorts in S', and *behaviour-definition* defines the functions h_s for all sorts $s \in S \cup S'$ excluding those in S for which no new terms arise. In all the examples below, it will be the case that the resulting specification is equivalent to a behaviour specification $h' : T_{\Sigma'}(A') \to A'$ where for each sort s in Σ', A'_s and h'_s are either inherited from h or defined explicitly in *sort-definitions* and *behaviour-definition*, respectively, and moreover, if h'_s is defined in *behaviour-definition* then it extends h_s. In fact, such a behaviour specification may be defined explicitly referring to the operations introduced by Def. 3.9, similarly as the standard enrich operation is defined in terms of translation and union [ST88a].

4 Observable behaviour

We now begin to focus on the main theme of this work, and look at what happens when we consider only *observable* behaviour. As usual, we regard values of some sorts as directly observable while the remaining sorts are treated as internal, with properties of their elements made visible only via observations, which are terms producing a result of observable sort. However, the technical means used to achieve this are somewhat different, at least superficially, from much previous work in this area. Without loss of generality, we take *bool* to be the only observable sort. In view of our earlier discussion on the role of *bool*, this means that we observe only the results of predicate applications, and take none of the ordinary "data" sorts as observable. This departs from standard approaches (a recent exception being [BST08]), where choosing a non-empty set of observable data sorts is crucial to have any observations at all and it is appropriate for this set to vary in the process of modular development (e.g. the parameter sorts in specifications of local constructions must be locally considered as observable). The former is taken care of by assuming that appropriate predicates are introduced into the specifications considered; the latter will be achieved in a technically different way here, see Def. 7.7 below.

Definition 4.1. *An* observable Σ-behaviour *is a function* $ev\colon (T_\Sigma)_{bool} \to \mathbb{B}$. *The* observable part *of a Σ-behaviour* $ev\colon T_\Sigma(A) \to A$ *is the restriction of* ev_{bool} *to* $(T_\Sigma)_{bool}$, *written as* $Obs(ev)\colon (T_\Sigma)_{bool} \to \mathbb{B}$.

The domain of the observable behaviour $Obs(ev)$ is properly included in the domain of the *bool* component of the behaviour ev: the latter also includes observations on non-ground terms, as well as the constants *true* and *false*. Including *true* and *false* would do no harm, but it is inappropriate to regard observations on unreachable values of non-observable sort as relevant to observable behaviour.

When producing a specification, we are actually interested in specifying *observable* behaviour; what happens with non-observable components is of no interest, except insofar as they affect the observable behaviour. The definition of observable behaviour specification is analogous to the definition of behaviour specification above and is inspired by the notion of "specification" in [GGM76], which is highlighted as their key definition.

Definition 4.2. *An* observable Σ-behaviour specification *is a function* $h\colon (T_\Sigma)_{bool} \to \mathbb{B} \cup \{\alpha\}$. *An observable Σ-behaviour* $ev\colon (T_\Sigma)_{bool} \to \mathbb{B}$ satisfies h *if* $ev(t) = h(t)$ *whenever* $h(t) \neq \alpha$. *A Σ-behaviour* $ev\colon T_\Sigma(A) \to A$ (observationally) satisfies h *if its observable part* $Obs(ev)\colon (T_\Sigma)_{bool} \to \mathbb{B}$ *satisfies* h. *We write* $Mod_{Obs}(h)$ *for the class of all such behaviours. Finally, the* observable part *of a Σ-behaviour specification* $h\colon T_\Sigma(A) \to A \cup \{\alpha\}$ *is the restriction of* h *to the set* $(T_\Sigma)_{bool}$; *we write this as* $Obs(h)\colon (T_\Sigma)_{bool} \to \mathbb{B} \cup \{\alpha\}$.

Lemma 4.3. *Let* $h\colon T_\Sigma(A) \to A \cup \{\alpha\}$ *be a Σ-behaviour specification. Then* $Mod(h) \subseteq Mod_{Obs}(Obs(h))$, *that is, for all Σ-behaviours ev, if ev satisfies h then ev observationally satisfies $Obs(h)$.*

Any observable Σ-behaviour specification $h\colon (T_\Sigma)_{bool} \to \mathbb{B} \cup \{\alpha\}$ may be equivalently considered as the behaviour specification $h^+\colon T_\Sigma(\mathbb{B}^+) \to \mathbb{B}^+ \cup \{\alpha\}$ that adds empty carriers for all sorts other than *bool* and maps all terms of these sorts to α.

Lemma 4.4. *Let* $h\colon (T_\Sigma)_{bool} \to \mathbb{B} \cup \{\alpha\}$ *be an observable Σ-behaviour specification. Then* $Mod_{Obs}(h) = Mod(h^+)$, *that is, for all Σ-behaviours ev, ev observationally satisfies h iff ev satisfies h^+.*

Any notion of observable behaviour gives rise to an equivalence between behaviours, whereby two behaviours are equivalent iff their observable parts coincide. This equivalence, methods for proving it, and conditions under which it is preserved by constructions on behaviours, is central to the study of observability concepts in specifications.

Definition 4.5. *Two Σ-behaviours, $ev\colon T_\Sigma(A) \to A$ and $ev'\colon T_\Sigma(A') \to A'$, are* observationally equivalent*, written $ev \equiv ev'$, if $Obs(ev) = Obs(ev')$.*

The following definition, and its use in the sequel, is derived from Schoett's notion of correspondence for Σ-algebras in [Sch87].

Definition 4.6. *Let $ev\colon T_\Sigma(A) \to A$ and $ev'\colon T_\Sigma(A') \to A'$ be Σ-behaviours. A* Σ-correspondence *$\rho\colon ev \bowtie ev'$ is an S-sorted relation $\rho \subseteq A \times A'$ such that ρ_{bool} is the identity relation on $\mathbb{B}$, and ρ is preserved by operations: for any operation name $f\colon s_1 \times \cdots . \times s_n \to s$ and terms $t_1, \ldots, t_n \in T_\Sigma(A)$ and $t'_1, \ldots, t'_n \in T_\Sigma(A')$ of the respective argument sorts, if $ev(t_1)\ \rho_{s_1}\ ev'(t'_1)$ and $\cdots$ and $ev(t_n)\ \rho_{s_n}\ ev'(t'_n)$ then $ev(f(t_1, \ldots, t_n))\ \rho_s\ ev'(f(t'_1, \ldots, t'_n))$.*

In fact, it is enough to consider here $t_1, \ldots, t_n$ and $t'_1, \ldots, t'_n$ to be constants in A and A' respectively.

Σ-correspondences can also be presented as spans of Σ-behaviour homomorphisms, see e.g. [BST08].

Proposition 4.7. *Let $ev\colon T_\Sigma(A) \to A$ and $ev'\colon T_\Sigma(A') \to A'$ be Σ-behaviours. Then $ev \equiv ev'$ iff there is a Σ-correspondence $\rho\colon ev \bowtie ev'$.*

Proposition 4.8. *Consider any Σ-behaviour specification $h\colon T_\Sigma(A) \to A \cup \{\alpha\}$ and its observable part $Obs(h)\colon (T_\Sigma)_{bool} \to \mathbb{B} \cup \{\alpha\}$. For every Σ-behaviour $ev\colon T_\Sigma(B) \to B$ that observationally satisfies $Obs(h)$, there exists a Σ-behaviour $ev'\colon T_\Sigma(C) \to C$ such that ev' satisfies h, and $ev \equiv ev'$.*

5 Examples

The following examples illustrate the definitions above. Examples in the sequel will build on these and will provide further illustrations.

We give an observable behaviour specification of symbol tables for a programming language with block structure and local variable declarations. This builds on the following specification of identifiers as strings.

IDENT =
  eqsort *ident* = *string*
  opns "a", "b", ..., "any string you like", ... : *string*

The notion of *eqsort* is borrowed from Standard ML's "eqtypes" (equality types). Since *ident* is an eqsort, it comes with an implicit operation $=\colon ident \times ident \to bool$ such that $h^{\textsc{Ident}}_{bool}(i = j) = true$ iff i and j are identical strings. We allow ourselves to write $i = j$ below in place of $h^{\textsc{Ident}}_{bool}(i = j) = true$ for brevity.

We use an informal notation for extending observable behaviour specifications, with a meaning that is analogous to that defined above for the case of behaviour specifications. According to the following specification, a symbol table records identifiers without associating any information to them. Identifiers that are added after entering a block are forgotten once the end of the block is reached, since identifiers within the block are no longer in scope at that point.

$$
\begin{array}{l}
\textsc{Symtab} = \textsc{Ident}\ \mathbf{then} \\
\quad \begin{array}{ll}
\mathbf{sort} & \mathit{symtab} \\
\mathbf{opns} & \mathit{empty}\colon \mathit{symtab} \\
 & \mathit{add}\colon \mathit{ident} \times \mathit{symtab} \to \mathit{symtab} \\
 & \mathit{enter}\colon \mathit{symtab} \to \mathit{symtab} \\
 & \mathit{leave}\colon \mathit{symtab} \to \mathit{symtab} \\
 & \mathit{isin}\colon \mathit{ident} \times \mathit{symtab} \to \mathit{bool}
\end{array}
\end{array}
$$

with

$$
\begin{array}{rl}
h^{\textsc{Symtab}}_{\mathit{bool}}(\mathit{isin}(i, \mathit{empty})) = & \mathit{false} \\
h^{\textsc{Symtab}}_{\mathit{bool}}(\mathit{isin}(i, \mathit{add}(i', t))) = & \alpha\ \mathbf{if}\ t \text{ is unbalanced} \\
 & \mathbf{else}\ \mathit{true}\ \mathbf{if}\ i = i' \\
 & \qquad \mathbf{else}\ h^{\textsc{Symtab}}_{\mathit{bool}}(\mathit{isin}(i, t)) \\
h^{\textsc{Symtab}}_{\mathit{bool}}(\mathit{isin}(i, \mathit{enter}(t))) = & \alpha\ \mathbf{if}\ t \text{ is unbalanced} \\
 & \mathbf{else}\ h^{\textsc{Symtab}}_{\mathit{bool}}(\mathit{isin}(i, t)) \\
h^{\textsc{Symtab}}_{\mathit{bool}}(\mathit{isin}(i, \mathit{leave}(t))) = & \alpha\ \mathbf{if}\ \mathit{leave}(t) \text{ is unbalanced} \\
 & \mathbf{else}\ h^{\textsc{Symtab}}_{\mathit{bool}}(\mathit{isin}(i, L(t)))
\end{array}
$$

where a (ground) term t is *unbalanced* if there is a subterm of t containing more occurrences of *leave* than *enter*. Informally, such a t is erroneous in the sense that it indicates an attempt to exit a block that has not been entered. Otherwise we say that t is *well-balanced*. Then L is an auxiliary function that for each t such that $\mathit{leave}(t)$ is well-balanced yields the term immediately inside the first use of *enter* that is not matched by a preceding *leave* (formally: $L(\mathit{add}(i, t)) = L(t)$, $L(\mathit{enter}(t)) = t$, and $L(\mathit{leave}(t)) = L(L(t))$). The meaning of an equation like $h^{\textsc{Symtab}}_{\mathit{bool}}(\mathit{isin}(i, \mathit{empty})) = \mathit{false}$ above is that it holds for all of its ground instances. (That is the reason why we needed constants in Ident.)

Symtab is then an observable behaviour specification over the indicated signature (i.e., the signature of Ident, including $=$, together with the sort *symtab* and the operations listed above).

A refinement of Symtab is Symtab′, given by the following function, which specifies choices for the cases that Symtab leaves open. It makes no use of α, and so it determines a single observable behaviour; all of the behaviours that satisfy it are observationally equivalent. We have decided here that for unbalanced terms t, $\mathit{isin}(i, t)$ yields true for any identifier i.

$$
\begin{aligned}
h_{bool}^{\textsc{Symtab}'}(isin(i, empty)) &= \mathit{false} \\
h_{bool}^{\textsc{Symtab}'}(isin(i, add(i', t))) &= \mathit{true}\ \textbf{if}\ i = i' \\
&\quad\ \textbf{else}\ h_{bool}^{\textsc{Symtab}'}(isin(i, t)) \\
h_{bool}^{\textsc{Symtab}'}(isin(i, enter(t))) &= h_{bool}^{\textsc{Symtab}'}(isin(i, t)) \\
h_{bool}^{\textsc{Symtab}'}(isin(i, leave(t))) &= \mathit{true}\ \textbf{if}\ leave(t)\ \text{is unbalanced} \\
&\quad\ \textbf{else}\ h_{bool}^{\textsc{Symtab}'}(isin(i, L(t)))
\end{aligned}
$$

Examples of behaviours over the signature of SYMTAB are given below in the form of SML structures. When no partiality, exceptions, polymorphism etc. arises, as below, this amounts to a definition of an algebra and therefore of a behaviour in our sense. Given such a structure definition STR, we will write STR.s for its carrier of sort s, and $ev^{\textsc{Str}}$ for its behaviour function.

The definitions below build on the definition of *ident* as the eqsort (or in SML, eqtype) *string* with constants as above.

```
structure LST =
  struct
    type symtab = (ident list) list
    val empty = [[]] : (ident list) list
    fun add(i,[]) = []
      | add(i,l::st) = (i::l)::st
    fun enter [] = []
      | enter st = []::st
    fun leave [] = []
      | leave(l::st) = st
    fun isin(i,[]) = false
      | isin(i,[]::st) = isin(i,st)
      | isin(i,(j::l)::st) = (i=j) orelse isin(i,l::st)
  end
```

Here, symbol tables are represented as lists of lists of identifiers. A list of identifiers represents the set of identifiers declared in a given block. A list of these lists is used to record block structure; this works because of the way that blocks can be nested. The behaviour determined by LST satisfies SYMTAB but does not satisfy SYMTAB′.

A different behaviour LST′ is obtained by making the *isin* function yield *true* for unbalanced symbol tables. In the code for LST, we just replace the definition of *isin* by

```
fun isin(i,[]) = true
  | isin(i,[[]]) = false
  | isin(i,[]::st) = isin(i,st)
  | isin(i,(j::l)::st) = (i=j) orelse isin(i,l::st)
```

(Note that the order of clauses matters in SML: the third clause only applies to non-empty *st*.) LST′ satisfies SYMTAB′ and so also satisfies SYMTAB.

A different behaviour with the same observable part as LST is given by the following structure, in which symbol tables are represented using functions. The set of identifiers within a given block is represented by its characteristic function, of type *ident* $\rightarrow$ *bool*, and a stack of these (represented as an "array" of sets, *int* $\rightarrow$ (*ident* $\rightarrow$ *bool*) together with an integer "pointer" to the top of the stack) is used to record block structure.

```
structure SST =
  struct
    type symtab = int -> (ident -> bool) * int
    val empty = (fn n => if n=0 then (fn i => false)
                         else (fn i => true) ,
                         0)
    fun add(i,(st,m)) =
          (fn n => if n=m
                   then fn j => (i=j) orelse st(n)(j)
                   else st(n) ,
           m)
    fun enter(st,m) =
          if m<0 then (st,m)
          else (fn n => if n>m then fn j=>false else st(n) ,
                m+1)
    fun leave(st,m) = (st,m-1)
    fun isin(i,(st,m)) = if m<0 then false
                         else if m=0 then st(m)(i)
                              else st(m)(i) orelse isin(i,(st,m-1))
  end
```

Now, LST and SST are observationally equivalent but there is no homomorphism from either to the other, even if we restrict their carriers to the values of ground terms. However, there are correspondences $\rho\colon ev^{\mathrm{SST}} \bowtie ev^{\mathrm{LST}}$ that witness the behavioural equivalence. One such correspondence relates all pairs $\langle st, m\rangle$ for $m < 0$ with the empty list, and then for $m \geq 0$ it relates $\langle st, m\rangle$ with all lists $[l_0, \ldots, l_m]$ such that $st(i)(j) = \mathit{true}$ iff j occurs at least once in l_i.

6 Implementations

We write specifications because we are interested in developing programs that implement them. One way of proceeding is top-down, by stepwise refinement: we refine the original specification of requirements to another one that is easier to implement by filling in design decisions such as choosing between the options of behaviour left open in "don't care" cases.

The issue of implementing specifications by programs is not mentioned in [GGM76]. An elegant approach to this issue has been developed in the years since then, coping with both observational and non-observational views of specifications. In this section and the next one we adapt this existing approach to the present framework, using our own work [ST88b,BST02,BST08] as a basis.

To produce a program from a specification, we proceed in stages by reducing the problem to a simpler one. At each stage, we postulate a solution to the simpler problem, and show by construction how to turn such a solution into a solution to the overall problem.

Definition 6.1. *Let Σ and Σ' be signatures. A* construction from Σ to Σ', *written $\kappa\colon \Sigma \Rightarrow \Sigma'$, is a function mapping any Σ-behaviour to a Σ'-behaviour.*[2] *Given a Σ-behaviour specification $h\colon T_\Sigma(A) \to A \cup \{\alpha\}$ and a Σ'-behaviour specification $h'\colon T_{\Sigma'}(A') \to A' \cup \{\alpha\}$, we say that h* implements h' via κ, *written $h' \underset{\kappa}{\rightsquigarrow} h$, if κ maps each Σ-behaviour satisfying h to a Σ'-behaviour satisfying h', i.e. $\kappa(Mod(h)) \subseteq Mod(h')$, where $\kappa(Mod(h))$ denotes the image of $Mod(h)$ under κ. When we want to emphasize correctness of κ in relating h and h' rather than the relationship between h and h', we say that κ* is correct w.r.t. h and h'.

Although the definition says that a construction $\kappa\colon \Sigma \Rightarrow \Sigma'$ is a mathematical function, it is best viewed as the semantic function underlying a *parameterised program* [Gog96], or in SML terms a *functor*, which produces the components (sorts and operations) required by Σ' when supplied with the components required by Σ. Then $h' \underset{\kappa}{\rightsquigarrow} h$ amounts to a reduction of the task of implementing h' to the task of implementing h, where κ supplies code to fill in the gap.

We can easily compose successive implementations. Then, once we have reduced the problem to one we have already solved, we obtain a solution to the original problem.

Proposition 6.2. *If $h_1 \underset{\kappa_1}{\rightsquigarrow} h_2 \underset{\kappa_2}{\rightsquigarrow} h_3$ then $h_1 \underset{\kappa_2;\kappa_1}{\rightsquigarrow} h_3$. Thus, if $h_1 \underset{\kappa_1}{\rightsquigarrow} \cdots \underset{\kappa_{n-1}}{\rightsquigarrow} h_n$ and ev_n satisfies h_n, then $\kappa_1(\cdots(\kappa_{n-1}(ev_n))\cdots)$ satisfies h_1.*

If we regard each construction as supplying some code, then composing a chain of constructions combines all of these program fragments into a single program.

This picture can be considerably enhanced to accommodate architectural system design [AG97] using multi-argument constructions to combine smaller components into a larger system — see [SST92,BST02].

A more sophisticated version of Def. 6.1 is needed to deal with the distinction between ordinary and observable behaviours. Since only observable aspects should determine correctness of implementations, in an implementation step $h' \underset{\kappa}{\rightsquigarrow} h$ it is too restrictive to require κ to deliver behaviours that "strictly" satisfy h'. We weaken this to satisfaction of only the observable part of h'.

Definition 6.3. *Given a Σ-behaviour specification $h\colon T_\Sigma(A) \to A \cup \{\alpha\}$ and a Σ'-behaviour specification $h'\colon T_{\Sigma'}(A') \to A' \cup \{\alpha\}$, we say that h* observationally implements h' via κ, *written $h' \overset{Obs}{\underset{\kappa}{\rightsquigarrow}} h$, if κ maps each Σ-behaviour satisfying h to a Σ'-behaviour satisfying $Obs(h')$, i.e. $\kappa(Mod(h)) \subseteq Mod_{Obs}(Obs(h'))$. Again, when we want to emphasize correctness of κ in relating h and h', we say that κ* is observationally correct w.r.t. h and h'.

This definition may be phrased in terms of observational equivalence: $h' \overset{Obs}{\underset{\kappa}{\rightsquigarrow}} h$ if for every Σ-behaviour ev satisfying h, there is a Σ'-behaviour ev' satisfying h' such that $\kappa(ev) \equiv ev'$. By Lemma 4.3, if $h' \underset{\kappa}{\rightsquigarrow} h$ then $h' \overset{Obs}{\underset{\kappa}{\rightsquigarrow}} h$.

[2] Constructions involved in practical examples may turn out to be *partial functions*; this may be dealt with similarly as in [BST02,BST08], so we disregard this issue here for the sake of simplicity.

But now composition of correctness is not so straightforward! Since observationally correct constructions build results that observationally satisfy the result specification given arguments that satisfy the argument specification "strictly", the following additional property is required to ensure that no problems arise when constructions are composed.

Definition 6.4. *A construction* $\kappa\colon \Sigma \Rightarrow \Sigma'$ *is* stable *if it preserves observational equivalence, that is, for all* Σ*-behaviours* ev_1 *and* ev_2*,* $ev_1 \equiv ev_2$ *implies* $\kappa(ev_1) \equiv \kappa(ev_2)$*.*

Proposition 6.5. *If* $h_1 \overset{Obs}{\underset{\kappa_1}{\leadsto}} h_2 \overset{Obs}{\underset{\kappa_2}{\leadsto}} h_3$ *and* κ_1 *is stable then* $h_1 \overset{Obs}{\underset{\kappa_2;\kappa_1}{\leadsto}} h_3$*.*

This suggests that in order to compose observational implementations, we must check that constructions are stable as well as checking that the implementing specification of one matches the implemented specification of the other. The definition of stability of $\kappa\colon \Sigma \Rightarrow \Sigma'$ involves quantification over all pairs of Σ-behaviours, so that could be difficult. But when constructions are determined by parameterised programs in a programming language, e.g. functors in SML, then it is possible to shift the burden of proof to the programming language designers by requiring that all *expressible* constructions be stable. This is entirely reasonable since it corresponds to requiring that parameterised programs respect abstraction boundaries: κ may freely use the components of its parameter that are listed in Σ, but may not take advantage of their particular internal properties.

6.1 Examples

A possible implementation of symbol tables as specified in Sect. 5, in terms of identifiers, proceeds in three steps. We implement Symtab by LBunch, then LBunch by Bunch, and finally Bunch by Ident. The intermediate specifications Bunch and LBunch are as follows.

$$
\begin{array}{l}
\textsc{Bunch} = \textsc{Ident}\ \textbf{then} \\
\quad \textbf{sort}\ \ bunch \\
\quad \textbf{opns}\ \ emptybunch\colon bunch \\
\quad \phantom{\textbf{opns}}\ \ defaultbunch\colon bunch \\
\quad \phantom{\textbf{opns}}\ \ addid\colon ident \times bunch \to bunch \\
\quad \phantom{\textbf{opns}}\ \ isinbunch\colon ident \times bunch \to bool \\
\quad \textbf{with} \\
\quad\quad bunch = ident\ list
\end{array}
$$

and

$$
\begin{array}{rl}
h^{\textsc{Bunch}}_{bunch}(emptybunch) = & \texttt{[]} \\
h^{\textsc{Bunch}}_{bunch}(defaultbunch) = & \alpha \\
h^{\textsc{Bunch}}_{bunch}(addid(i,b)) = & \alpha\ \textbf{if}\ h^{\textsc{Bunch}}_{bunch}(b) = \alpha \\
 & \textbf{else}\ i::h^{\textsc{Bunch}}_{bunch}(b) \\
h^{\textsc{Bunch}}_{bool}(isinbunch(i,b)) = & \alpha\ \textbf{if}\ h^{\textsc{Bunch}}_{bunch}(b) = \alpha \\
 & \textbf{else case}\ h^{\textsc{Bunch}}_{bunch}(b)\ \textbf{of}\ \texttt{[]} \quad \Rightarrow false \\
 & \qquad\qquad\qquad\qquad\quad j::b' \Rightarrow true\ \textbf{if}\ i = j \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ isinbunch(i,b')
\end{array}
$$

$\text{LB}\textsc{unch} = \textsc{Bunch}$ **then**

$$
\begin{array}{ll}
\textbf{sort} & \mathit{lb} \\
\textbf{opns} & \mathit{emptylb}\colon \mathit{lb} \\
& \mathit{addbunch}\colon \mathit{bunch} * \mathit{lb} \to \mathit{lb} \\
& \mathit{popbunch}\colon \mathit{lb} \to \mathit{lb} \\
& \mathit{topbunch}\colon \mathit{lb} \to \mathit{bunch} \\
& \mathit{isemptylb}\colon \mathit{lb} \to \mathit{bool} \\
\textbf{with} & \\
\quad \mathit{lb} = \mathit{bunch\ list} &
\end{array}
$$

and

$$
\begin{aligned}
h^{\text{LB}\textsc{unch}}_{\mathit{lb}}(\mathit{emptylb}) &= \texttt{[]} \\
h^{\text{LB}\textsc{unch}}_{\mathit{lb}}(\mathit{addbunch}(b,l)) &= \alpha \textbf{ if } h^{\text{LB}\textsc{unch}}_{\mathit{lb}}(l) = \alpha \textbf{ or } h^{\text{LB}\textsc{unch}}_{\mathit{bunch}}(b) = \alpha \\
&\quad \textbf{else } b :: h^{\text{LB}\textsc{unch}}_{\mathit{lb}}(l) \\
h^{\text{LB}\textsc{unch}}_{\mathit{lb}}(\mathit{popbunch}(l)) &= \alpha \textbf{ if } h^{\text{LB}\textsc{unch}}_{\mathit{lb}}(l) = \alpha \\
&\quad \textbf{else case } h^{\text{LB}\textsc{unch}}_{\mathit{lb}}(l) \textbf{ of } \texttt{[]} \Rightarrow \alpha \\
&\qquad\qquad\qquad\qquad\quad\ b :: l' \Rightarrow l' \\
h^{\text{LB}\textsc{unch}}_{\mathit{bunch}}(\mathit{topbunch}(l)) &= \alpha \textbf{ if } h^{\text{LB}\textsc{unch}}_{\mathit{lb}}(l) = \alpha \\
&\quad \textbf{else case } h^{\text{LB}\textsc{unch}}_{\mathit{lb}}(l) \textbf{ of } \texttt{[]} \Rightarrow \alpha \\
&\qquad\qquad\qquad\qquad\quad\ b :: l' \Rightarrow b \\
h^{\text{LB}\textsc{unch}}_{\mathit{bool}}(\mathit{isemptylb}(l)) &= \alpha \textbf{ if } h^{\text{LB}\textsc{unch}}_{\mathit{lb}}(l) = \alpha \\
&\quad \textbf{else case } h^{\text{LB}\textsc{unch}}_{\mathit{lb}}(l) \textbf{ of } \texttt{[]} \Rightarrow \mathit{true} \\
&\qquad\qquad\qquad\qquad\quad\ b :: l' \Rightarrow \mathit{false} \\
h^{\text{LB}\textsc{unch}}_{\mathit{bunch}}(\mathit{addid}(i,b)) &= \alpha \textbf{ if } h^{\text{LB}\textsc{unch}}_{\mathit{bunch}}(b) = \alpha \\
&\quad \textbf{else } i :: h^{\text{LB}\textsc{unch}}_{\mathit{bunch}}(b) \\
h^{\text{LB}\textsc{unch}}_{\mathit{bool}}(\mathit{isinbunch}(i,b)) &= \alpha \textbf{ if } h^{\text{LB}\textsc{unch}}_{\mathit{bunch}}(b) = \alpha \\
&\quad \textbf{else case } h^{\text{LB}\textsc{unch}}_{\mathit{bunch}}(b) \\
&\qquad\qquad \textbf{of } \texttt{[]} \Rightarrow \mathit{false} \\
&\qquad\qquad\quad j :: b' \Rightarrow \mathit{true} \textbf{ if } i = j \\
&\qquad\qquad\qquad\qquad \textbf{else } \mathit{isinbunch}(i,b')
\end{aligned}
$$

Bunch describes an abstract data type for a collection of identifiers with membership. LBunch, short for "layered bunch", comes from the recognition that a stack-like structure is relevant to dealing with entering and leaving blocks.

Note that the last two clauses of LBunch need to be included, although they are essentially repeated from Bunch: here b ranges over a larger set of terms. A full-blown specification language for writing specifications in this style would include notational conventions for circumventing this kind of boring and error-prone repetition, as well as other inconveniences of the notation above.

Now we give the constructions, which are functions from behaviours to behaviours, in a number of variants, as SML functors. We proceed bottom-up, starting with two variants of the implementation of Bunch by Ident.

```
functor FB(structure I : IDENT) : BUNCH =
  struct
    open I
    type bunch = ident list
```

```
    val emptybunch = []
    val defaultbunch = []
    fun addid(i,b) = i::b
    fun isinbunch(i,[]) = false
      | isinbunch(i,j::b) = (i=j) orelse isinbunch(i,b)
  end
```

Clearly, for any behaviour Id that satisfies Ident, Fb(Id) yields a behaviour that satisfies Bunch, and so Fb (the semantic function underlying the functor FB) is correct w.r.t. Ident and Bunch, i.e. Bunch $\underset{\mathrm{Fb}}{\rightsquigarrow}$ Ident.

```
functor FB'(structure I : IDENT) : BUNCH =
  struct
    open I
    type bunch = ident -> bool
    val emptybunch = fn i => false
    val defaultbunch = fn i => true
    fun addid(i,f) = fn j => (i=j) orelse f(j)
    fun isinbunch(i,f) = f(i)
  end
```

Given any behaviour Id that satisfies Ident, Fb'(Id) yields a behaviour that does not satisfy Bunch. However, it does observationally satisfy the observable part of Bunch and so Fb' is observationally correct w.r.t. Ident and Bunch, i.e. Bunch $\overset{Obs}{\underset{\mathrm{Fb}'}{\rightsquigarrow}}$ Ident.

Now we consider two ways of implementing LBunch by Bunch.

```
functor FLB(structure B : BUNCH) : LBUNCH =
  struct
    open B
    type lb = bunch list
    val emptylb = []
    fun addbunch(b,l) = b::l
    fun popbunch [] = []
      | popbunch(b::l) = l
    fun topbunch [] = defaultbunch
      | topbunch(b::l) = b
    fun isemptylb [] = true
      | isemptylb(b::l) = false
  end
```

For any behaviour B that satisfies Bunch, Flb(B) yields a behaviour that satisfies LBunch, and thus LBunch $\underset{\mathrm{Flb}}{\rightsquigarrow}$ Bunch.

```
functor FLB'(structure B : BUNCH) : LBUNCH =
  struct
    open B
    type lb = (int -> bunch) * int
    val emptylb = (fn n => defaultbunch , ~1)
    fun addbunch(b,(f,m)) = (fn n => if n>m then b else f(n) , m+1)
```

```
    fun popbunch(f,m) = (f,m-1)
    fun topbunch(f,m) = f(m)
    fun isemptylb(f,m) = m<0
  end
```

For any behaviour B that satisfies Bunch, $\text{Flb}'(B)$ yields a behaviour that satisfies the observable part of LBunch, and so LBunch $\underset{\text{Flb}'}{\overset{Obs}{\rightsquigarrow}}$ Bunch.

Finally, we give two ways of implementing Symtab by LBunch.

```
functor FST(structure LB : LBUNCH) : SYMTAB =
  struct
    eqtype ident = LB.ident
    val "a" = LB."a"   val "b" = LB."b"   ...
    type symtab = LB.lb
    val empty = LB.addbunch(LB.emptybunch,LB.emptylb)
    fun add(i,st) =
          if LB.isemptylb(st) then st
          else LB.addbunch(LB.addid(i,LB.topbunch(st)),LB.popbunch(st))
    fun enter(st) = if LB.isemptylb(st) then st
                    else LB.addbunch(LB.emptybunch,st)
    val leave = LB.popbunch
    fun isin(i,st) =
          if LB.isemptylb(st) then false
          else if LB.isemptylb(LB.popbunch(st))
               then LB.isinbunch(i,LB.topbunch(st))
               else LB.isinbunch(i,LB.topbunch(st))
                    orelse LB.isin(i,LB.popbunch(st))
  end

functor FST'(structure LB : LBUNCH) : SYMTAB =
  struct
    eqtype ident = LB.ident
    val "a" = LB."a"   val "b" = LB."b"   ...
    type symtab = LB.lb
    val empty = LB.addbunch(LB.emptybunch,LB.emptylb)
    fun add(i,st) =
          if LB.isemptylb(st) then st
          else LB.addbunch(LB.addid(i,LB.topbunch(st)),LB.popbunch(st))
    fun enter(st) =
          if LB.isemptylb(st) then st
          else LB.addbunch(LB.emptybunch,st)
    val leave = LB.popbunch
    fun isin(i,st) =
          LB.isemptylb(st) orelse
          LB.isinbunch(i,LB.topbunch(st)) orelse
          LB.isin(i,LB.popbunch(st))
  end
```

For any behaviour Lb that satisfies LBunch, each of Fst(Lb) and $\text{Fst}'(\text{Lb})$ yields a behaviour that satisfies Symtab, and thus we have Symtab $\underset{\text{Fst}}{\rightsquigarrow}$

LBunch and Symtab $\underset{\text{FST}'}{\rightsquigarrow}$ LBunch. And so, for each of these, we also have an observational implementation, Symtab $\overset{Obs}{\underset{\text{FST}}{\rightsquigarrow}}$ LBunch etc.

All of these constructions are stable; since they are coded as closed SML functors, under the very plausible conjecture that all closed SML-expressible functors are stable. (Actually proving this result would be very tedious, as the proof would need to consider the entire definition of SML. But it would only need to be done once, thereafter freeing implementors from the obligation to check stability case by case.) Consequently, the obvious compositions of these constructions yield behaviours that observationally satisfy Symtab. In particular, for any behaviour Id that satisfies Ident, Fst(Flb(Fb(Id))) corresponds to Lst, Fst$'$(Flb(Fb(Id))) corresponds to Lst$'$ and Fst(Flb$'$(Fb$'$(Id))) corresponds to Sst. Other combinations, like Fst(Flb$'$(Fb(Id))) and Fst$'$(Flb(Fb$'$(Id))), yield still different structures.

Now consider the following modification to FST:

```
functor FSTBAD(structure LB : LBUNCH) : SYMTAB =
  struct
    eqtype ident = LB.ident
    ...
    fun isin(i,st) =
          if (st = LB.emptylb) then false
          else if LB.popbunch(st) = LB.emptylb
               then LB.isinbunch(i,LB.topbunch(st))
               else LB.isinbunch(i,LB.topbunch(st))
                    orelse LB.isin(i,LB.popbunch(st))
  end
```

(This is not valid in SML: the type `LB.lb` is not required to admit equality and so `st = LB.emptylb` does not typecheck.)

FstBad is still observationally correct: for every behaviour that satisfies LBunch, FstBad builds a behaviour that observationally satisfies Symtab (since for each Lb satisfying LBunch, for all well-balanced terms t of type *symtab*, the values of $isin(i, t)$ in FstBad(Lb) and in Fst(Lb) coincide). But FstBad is not stable. So, if we take Flb$'$(Fb$'$(Id)), which does not satisfy LBunch (even though it satisfies the observable part of it), and then apply FstBad to build Bad = FstBad(Flb$'$(Fb$'$(Id))), Bad need not satisfy Symtab. Indeed, it does not: evaluating the term $isin(j, leave(add(i, enter(empty))))$ in Bad gives *true*, and this is incorrect according to Symtab.

7 Local constructions in global contexts

Very informally, constructions as discussed in Sect. 6 were considered at the "global" level of the entire system under development: they build a behaviour that implements the overall requirements specification given any argument behaviour satisfying a specification of the part of the system yet to be implemented. But in each development step the construction typically uses only a relatively small part of its argument to add a new part of the result, and just passes most of the argument over to the result without using or touching it in any way.

We will now have a closer look at one aspect of modular development, namely at the use of *local* constructions, which take as argument only as much of the behaviour as is necessary to build some new part of the result. Such local constructions give rise to constructions at the "global" level, whereby the required argument is cut out of the global context and the result combined with the same global context, thus extending it by new parts and contributing to the overall system implementation. We capture this idea using the pushout technique and amalgamation, as is standard in algebraic specifications [EM85].

A technical tool we need is an extension of Lemma 2.3 to behaviours and correspondences.

Lemma 7.1. *Consider a pushout in the category of signatures.*

$$\begin{tikzcd} \Sigma_1 \arrow[r, "\sigma'_2"] & \Sigma' \\ \Sigma \arrow[u, "\sigma_1"] \arrow[r, "\sigma_2"'] & \Sigma_2 \arrow[u, "\sigma'_1"'] \end{tikzcd}$$

Then for any Σ_1-behaviour ev_1 and Σ_2-behaviour ev_2 with common Σ-reduct $ev_1|_{\sigma_1} = ev_2|_{\sigma_2}$, there exists a unique Σ'-behaviour ev' such that $ev'|_{\sigma'_2} = ev_1$ and $ev'|_{\sigma'_1} = ev_2$. Similarly for correspondences: for any Σ_1-correspondence $\rho_1\colon ev_{1,1} \bowtie ev_{1,2}$ and Σ_2-correspondence $\rho_2\colon ev_{2,1} \bowtie ev_{2,2}$ with common Σ-reducts $ev_{1,1}|_{\sigma_1} = ev_{2,1}|_{\sigma_2}$, $ev_{1,2}|_{\sigma_1} = ev_{2,2}|_{\sigma_2}$, and $\rho_1|_{\sigma_1} = \rho_2|_{\sigma_2}$, there exists a unique Σ'-correspondence $\rho'\colon ev'_1 \bowtie ev'_2$ such that $\rho'|_{\sigma'_2} = \rho_1$ and $\rho'|_{\sigma'_1} = \rho_2$, where $ev'_1|_{\sigma'_2} = ev_{1,1}$, $ev'_1|_{\sigma'_1} = ev_{2,1}$, $ev'_2|_{\sigma'_2} = ev_{1,2}$, $ev'_2|_{\sigma'_1} = ev_{2,2}$.

We are ready now to state the main definitions for this section:

Definition 7.2. *Given a signature morphism $\iota\colon \Sigma \to \Sigma'$, a* (local) construction along *ι is a function κ that maps any Σ-behaviour ev to a Σ'-behaviour $\kappa(ev)$ such that $\kappa(ev)|_\iota = ev$.*

$$\begin{tikzcd} \Sigma_G \arrow[r, "\iota'"] & \Sigma'_G \\ \Sigma \arrow[u, "\gamma"] \arrow[r, "\iota"'] & \Sigma' \arrow[u, "\gamma'"'] \end{tikzcd}$$

Then, given a ("global context") signature Σ_G and a ("fitting") morphism $\gamma\colon \Sigma \to \Sigma_G$, the construction κ^γ along $\iota'\colon \Sigma_G \to \Sigma'_G$ induced by κ via γ *(or the* γ-lifting κ^γ of κ*) is defined for any Σ_G-behaviour ev_G so that $\kappa^\gamma(ev_G)$ is the Σ'_G-behaviour such that $\kappa^\gamma(ev_G)|_{\iota'} = ev_G$ and $\kappa^\gamma(ev_G)|_{\gamma'} = \kappa(ev_G|_\gamma)$, where Σ'_G and morphisms $\iota'\colon \Sigma_G \to \Sigma'_G$, $\gamma'\colon \Sigma' \to \Sigma'_G$ are given by a pushout of $\iota\colon \Sigma \to \Sigma'$ and $\gamma\colon \Sigma \to \Sigma_G$.*

As argued in Sect. 6, the key property of (global) constructions is stability. Unfortunately, since enlarging the context of use typically expands observable behaviour, stability of a local construction will not ensure that its lifting along a fitting morphism is stable as well. Following [Sch87] we introduce a stronger property, which is preserved by lifting along any morphism.

Definition 7.3. *A construction κ along a signature morphism $\iota\colon \Sigma \to \Sigma'$ is* locally stable *if it extends correspondences, that is, for any Σ-behaviours ev_1 and ev_2, any Σ-correspondence $\rho\colon ev_1 \bowtie ev_2$ extends to a Σ'-correspondence $\rho'\colon \kappa(ev_1) \bowtie \kappa(ev_2)$ such that $\rho'|_\iota = \rho$.*

Theorem 7.4. *A construction κ along a signature morphism $\iota\colon \Sigma \to \Sigma'$ is locally stable if and only if for all signatures Σ_G and fitting morphisms $\gamma\colon \Sigma \to \Sigma_G$, the γ-lifting κ^γ of κ is stable.*

The "only if" part of this theorem is what we need: local stability of a local construction ensures its stability in any context of use; the "if" part shows that no weaker condition can be given.

We now turn to the issue of specifying local constructions and using their specifications to justify correctness of global implementation steps. The standard approach would be that to specify local constructions along $\iota'\colon \Sigma \to \Sigma'$, one gives a Σ-behaviour specification that determines the arguments intended for the construction, and a Σ'-behaviour specification that describes the results built. Unfortunately, this doesn't work in the framework discussed in this paper. Any behaviour specification constrains the behaviour only on elements that are in the specification's carrier, since behaviours which satisfy that specification are required to conform on such elements only, not on additional elements that may also be present in their carriers. When a local construction is used in a global context, the set of data for which the behaviour is of importance may grow beyond what we can take explicit account of when writing a single specification. We accommodate this by making the result specification depend on the carriers of the argument supplied.[3]

Definition 7.5. *A* specification of local constructions *along $\iota\colon \Sigma \to \Sigma'$ consists of a Σ-behaviour specification $h\colon T_\Sigma(A) \to A \cup \{\alpha\}$ and a function that maps any set $X \supseteq A$ to a Σ'-behaviour specification $h'_X : T_\Sigma(X') \to X' \cup \{\alpha\}$. We write such a specification as $\Pi X{:}h{\to}h'_X$.*

A local construction κ along $\iota\colon \Sigma \to \Sigma'$ strictly satisfies *(or,* is strictly correct w.r.t.*) $\Pi X{:}h{\to}h'_X$ if for any Σ-behaviour $ev\colon T_\Sigma(X) \to X$ that satisfies h, $\kappa(ev)$ is a Σ'-behaviour that satisfies h'_X.*

The conditions to ensure that a strictly correct local construction lifts to a strictly correct construction in a global context are now rather natural:

Theorem 7.6. *Let κ be a local construction along $\iota\colon \Sigma \to \Sigma'$ that strictly satisfies $\Pi X{:}h{\to}h'_X$.*

Consider a signature Σ_G with a fitting morphism $\gamma\colon \Sigma \to \Sigma_G$ and the usual pushout. Let $h_G\colon T_{\Sigma_G}(A_G) \to A_G \cup \{\alpha\}$ and $h'_G\colon T_{\Sigma'_G}(A'_G) \to A'_G \cup \{\alpha\}$ be behaviour specifications. If $\gamma(h)$ refines h_G, and for each $X \supseteq A_G|_\gamma$, h'_G refines $\iota'(h) + \gamma'(h'_X)$ then the γ-lifting κ^γ of κ is strictly correct w.r.t. h_G and h'_G.

$$\begin{tikzcd} \Sigma_G \arrow[r, "\iota'"] & \Sigma'_G \\ \Sigma \arrow[u, "\gamma"] \arrow[r, "\iota"'] & \Sigma' \arrow[u, "\gamma'"'] \end{tikzcd}$$

Let us now turn to observational correctness. As with stability, observational correctness for (global) constructions would be too weak for local constructions:

[3] Allowing, more generally, the result specification to depend on the entire argument behaviour, not just its carriers, should not cause extra technical difficulties.

when such a local construction is used in a global context, where more observations are available, correctness would be lost in general. The following definition strengthens the requirements appropriately:

Definition 7.7. *A local construction κ along $\iota\colon \Sigma \to \Sigma'$* locally satisfies *(or,* is locally correct w.r.t.*) $\Pi X{:}h{\to}h'_X$ if for any Σ-behaviour $ev\colon T_\Sigma(X) \to X$ that satisfies h, there is a Σ'-behaviour ev' that satisfies h'_X and a Σ'-correspondence $\rho'\colon ev' \bowtie \kappa(ev)$ such that $\rho'|_\iota$ is the identity (which implies $ev'|_\iota = ev$).*

Local correctness is stronger than observational correctness of κ, which would just state that $\kappa(ev)$ observationally satisfies h'_X. By requiring that this is "witnessed" by a correspondence that is identity on the argument sorts, we "locally" fix the argument sorts as observable, thus allowing arbitrary observations for them to be added in the context of use, as the following key theorem shows.

Theorem 7.8. *Let κ be a local construction along $\iota\colon \Sigma \to \Sigma'$ that is locally stable and locally satisfies $\Pi X{:}h{\to}h'_X$.*

Consider a signature Σ_G with a fitting morphism $\gamma\colon \Sigma \to \Sigma_G$ and the usual pushout. Let $h_G\colon T_{\Sigma_G}(A_G) \to A_G \cup \{\alpha\}$ and $h'_G\colon T_{\Sigma'_G}(A'_G) \to A'_G \cup \{\alpha\}$ be behaviour specifications. If $Mod(h_G) \subseteq Mod_{Obs}(Obs(h_G{+}\gamma(h)))$, and for each $X \supseteq A_G|_\gamma$, $Mod(\iota'(h) + \gamma'(h'_X)) \subseteq Mod_{Obs}(Obs(h'_G))$ then the γ-lifting κ^γ of κ is observably correct w.r.t. h_G and h'_G.

$$\begin{tikzcd}
\Sigma_G \arrow[r, "\iota'"] & \Sigma'_G \\
\Sigma \arrow[u, "\gamma"] \arrow[r, "\iota"'] & \Sigma' \arrow[u, "\gamma'"']
\end{tikzcd}$$

7.1 Examples

Recall constructions FLB and FLB$'$ as defined in Sect. 6. Any argument behaviour B (which has to satisfy BUNCH) is inherited by the result behaviours FLB(B) and FLB(B), but it is passed there untouched and otherwise (apart from the type *bunch* and the constant *defaultbunch*) is not used by FLB and FLB$'$. Therefore the essence of either of the two constructions can be captured over a smaller signature as a local construction as follows.

We start with a simple argument specification:

ELEM =
sort *elem*
opns *default*: *elem*
with
elem = *unit*
and
$h^{\text{ELEM}}_{elem}(default) = \alpha$

The following function yields specifications for the result given a carrier for sort *elem*; note that all these result specifications have a common signature.

$$
\begin{array}{l}
\textsc{Stack}(\textbf{sort}\ \mathit{elem}) = \\
\quad \textbf{sort}\ \ \mathit{stack} \\
\quad \textbf{opns}\ \mathit{emptystack} : \mathit{stack} \\
\qquad\qquad \mathit{push} : \mathit{elem} \times \mathit{stack} \to \mathit{stack} \\
\qquad\qquad \mathit{pop} : \mathit{stack} \to \mathit{stack} \\
\qquad\qquad \mathit{top} : \mathit{stack} \to \mathit{elem} \\
\qquad\qquad \mathit{isempty} : \mathit{stack} \to \mathit{bool} \\
\quad \textbf{with} \\
\qquad \mathit{stack} = \mathit{elem}\ \texttt{list}
\end{array}
$$

and

$$
\begin{array}{rl}
h^{\textsc{Stack}}_{\mathit{stack}}(\mathit{emptystack}) = & \texttt{[]} \\
h^{\textsc{Stack}}_{\mathit{stack}}(\mathit{push}(b,l)) = & \alpha\ \textbf{if}\ h^{\textsc{Stack}}_{\mathit{stack}}(l) = \alpha\ \textbf{or}\ h^{\textsc{Stack}}_{\mathit{elem}}(b) = \alpha \\
& \textbf{else}\ b :: h^{\textsc{Stack}}_{\mathit{stack}}(l) \\
h^{\textsc{Stack}}_{\mathit{stack}}(\mathit{pop}(l)) = & \alpha\ \textbf{if}\ h^{\textsc{Stack}}_{\mathit{stack}}(l) = \alpha \\
& \textbf{else case}\ h^{\textsc{Stack}}_{\mathit{stack}}(l)\ \textbf{of}\ \texttt{[]} \quad \Rightarrow \alpha \\
& \qquad\qquad\qquad\qquad\quad b :: l' \Rightarrow l' \\
h^{\textsc{Stack}}_{\mathit{elem}}(\mathit{top}(l)) = & \alpha\ \textbf{if}\ h^{\textsc{Stack}}_{\mathit{stack}}(l) = \alpha \\
& \textbf{else case}\ h^{\textsc{Stack}}_{\mathit{stack}}(l)\ \textbf{of}\ \texttt{[]} \quad \Rightarrow \alpha \\
& \qquad\qquad\qquad\qquad\quad b :: l' \Rightarrow b \\
h^{\textsc{Stack}}_{\mathit{bool}}(\mathit{isempty}(l)) = & \alpha\ \textbf{if}\ h^{\textsc{Stack}}_{\mathit{stack}}(l) = \alpha \\
& \textbf{else case}\ h^{\textsc{Stack}}_{\mathit{stack}}(l)\ \textbf{of}\ \texttt{[]} \quad \Rightarrow \mathit{true} \\
& \qquad\qquad\qquad\qquad\quad b :: l' \Rightarrow \mathit{false}
\end{array}
$$

Here are two local constructions along the obvious inclusion of the signature of Elem into the signature of Stack(*elem*) (for any carrier for *elem*).

```
functor FSTACK(structure E : ELEM) : STACK(E.elem) =
  struct
    open E
    type stack = elem list
    val emptystack = []
    fun push(b,l) = b::l
    fun pop [] = []
      | pop(b::l) = l
    fun top [] = default
      | top(b::l) = b
    fun isempty [] = true
      | isempty(b::l) = false
  end

functor FSTACK'(structure E : ELEM) : STACK(E.elem) =
  struct
    open E
    type stack = (int -> elem) * int
    val emptystack = (fn n => default , -1)
    fun push(b,(f,m)) = (fn n => if n>m then b else f(n), m+1)
    fun pop(f,m) = (f,m-1)
```

```
    fun top(f,m) = f(m)
    fun isempty(f,m) = m<0
  end
```

FSTACK and FSTACK′ are locally stable. As in Sect. 6.1, this should follow from the fact that they are coded as SML functors. FSTACK and FSTACK′ are also locally correct.

FLB and FLB′ arise as global constructions induced by FSTACK and FSTACK′, respectively, via the fitting morphism from the signature of ELEM to the signature of BUNCH which maps *elem* to *bunch* and *default* to *defaultbunch*. This is, of course, assuming the appropriate choice of names in the pushout signature, where we need to rename *stack* to *lb*, *emptystack* to *emptylb*, etc. Stability and observational correctness of FLB and FLB′ follow from local stability and local correctness of FSTACK and FSTACK′, respectively, using Thms. 7.4 and 7.8.

8 Conclusion

This is a new look at some of our previous work on observational interpretation of specifications and its role in software specification and development [ST88b,BST02,BST08], presented in a new framework inspired by the ideas in [GGM76]. We have focused on a key idea in [GGM76], that of viewing a system via its behaviour, given by the evaluation function for terms of sort *bool*, and consequently, that of specifying behaviour by indicating the results of evaluating some terms and leaving others as "don't care" cases. The resulting framework and its observational aspects are sketched in Sects. 3 and 4; our work on systematic software development is then adapted to this framework, concentrating on the use of local constructions in a global context, in Sects. 6 and 7.

The specifications considered in this new framework are considerably more restrictive than axiomatic specifications. Even the simple constraint that the values of two terms coincide cannot be captured without giving their common value explicitly. On the other hand, such specifications offer a visible link to so-called "abstract model specifications" [Jon80] with mechanisms for making looseness in specifications explicit, via the designation of "don't care" cases, rather than implicit, by simply omitting axioms from specifications that constrain required behaviour. The ramifications of this link remains to be investigated. It is possible to add axiomatic specifications to this framework, for which there are two approaches: one uses an observational interpretation of the axioms, where equality refers to indistinguishability via the available operations; the other uses the standard interpretation, but closes model classes of specifications under behavioural equivalence.

Other important parts of the story are not covered here for lack of space. One major issue concerns proof: to show that a behaviour satisfies a specification, or that one specification implements another, requires formal proofs about behaviours and specifications. An observational view of specifications complicates this task, although again requiring stability helps considerably. But this is still a research area, with some questions unresolved after many years of work. A recent elegant and promising approach to these issues is [BH06].

Another point concerns the extension of this account to deal with more complex notions of behaviour, involving partial functions, higher-order functions, subsorting, relations etc. When we depart from the total first-order case presented here, it is not always obvious what observable behaviour means: should partiality of operations be observable or not? Should the results of relations, as opposed to functions into *bool*, be observable or not?

Parallel to the developments in algebraic specifications, ideas concerning observability have been taken up in the area of concurrency (with significant contributions from Ugo Montanari in this area too!). Essentially from their beginnings, standard process calculi have been considered modulo a notion of bisimulation in various variants, which captures similar intuitions as that of observational equivalence studied in work on algebraic specifications. An abstract version of bisimulation, defined using spans of "open maps" [JNW96] can be used to link the two concepts, see [Las97,Las98]. It would be interesting to see how these ideas may be instantiated in the present framework.

Yet another angle on this topic is provided by *universal coalgebra* [Rut00], with techniques to specify coalgebras (and behaviours they define) using modal logics [Kur01]. We would like to try extending the approach presented here in this direction as well.

References

[AG97] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

[AKKB99] E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors. *Algebraic Foundations of Systems Specification*. Springer, 1999.

[BH06] M. Bidoit and R. Hennicker. Constructor-based observational logic. *Journal of Logic and Algebraic Programming*, 67(1–2):3–51, 2006.

[BKL+91] M. Bidoit, H.-J. Kreowski, P. Lescanne, F. Orejas, and (eds.) D. Sannella. *Algebraic System Specification and Development: A Survey and Annotated Bibliography*. Springer LNCS 501, 1991.

[BST02] M. Bidoit, D. Sannella, and A. Tarlecki. Architectural specifications in Casl. *Formal Aspects of Computing*, 13:252–273, 2002.

[BST08] M. Bidoit, D. Sannella, and A. Tarlecki. Observational interpretation of Casl specifications. *Mathematical Structures in Computer Science*, 18:325–371, 2008.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. Springer, 1985.

[GGM76] V. Giarratana, F. Gimona, and U. Montanari. Observability concepts in abstract data type specifications. In *Proc. 1976 Symp. on Mathematical Foundations of Computer Science*, pages 567–578. Springer LNCS 45, 1976.

[GM00] Joseph A. Goguen and Grant Malcolm. A hidden agenda. *Theoretical Computer Science*, 245(1):55–101, 2000.

[Gog96] J.A. Goguen. Parameterized programming and software architecture. In *Proc. 4th Intl. IEEE Conf. on Software Reuse*, pages 2–11, 1996.

[GTW78] J.A. Goguen, J.W. Thatcher, and E.G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In *Current Trends in Programming Methodology, Vol. 4: Data Structuring*, pages 80–149. Prentice-Hall, 1978. Edited by R.T. Yeh.

[JNW96] A. Joyal, M. Nielsen, and G. Winskel. Bisimulation from open maps. *Information and Computation*, 127(2):164–185, 1996.

[Jon80] C.B. Jones. *Software Development: A Rigorous Approach*. Prentice-Hall, 1980.

[Kur01] Alexander Kurz. Specifying coalgebras with modal logic. *Theoretical Computer Science*, 260(1-2):119–138, 2001.

[Las97] S. Lasota. Open maps as a bridge between algebraic observational equivalence and bisimilarity. In *Recent Trends in Algebraic Development Techniques*, pages 285–299. Springer LNCS 1376, 1997.

[Las98] S. Lasota. Partial-congruence factorization of bisimilarity induced by open maps. In *Proc. 25th Intl. Colloq. on Automata, Languages and Programming*, pages 91–102. Springer LNCS 1443, 1998.

[Rei81] H. Reichel. Behavioural equivalence – a unifying concept for initial and final specification methods. In *Proc. 3rd Hungarian Comp. Sci. Conference*, pages 27–39, 1981.

[Rut00] Jan J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249(1):3–80, 2000.

[Sch87] O. Schoett. *Data Abstraction and the Correctness of Modular Programming*. PhD thesis, Dept. of Computer Science, Univ. of Edinburgh, 1987.

[SST92] D. Sannella, S. Sokołowski, and A. Tarlecki. Toward formal development of programs from algebraic specifications: Parameterisation revisited. *Acta Informatica*, 29(8):689–736, 1992.

[ST87] D. Sannella and A. Tarlecki. On observational equivalence and algebraic specification. *Journal of Computer and System Sciences*, 34:150–178, 1987.

[ST88a] D. Sannella and A. Tarlecki. Specifications in an arbitrary institution. *Information and Computation*, 76:165–210, 1988.

[ST88b] D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: Implementations revisited. *Acta Informatica*, 25:233–281, 1988.

[Wan79] M. Wand. Final algebra semantics and data type extensions. *Journal of Computer and System Sciences*, 19:27–44, 1979.

[Wir90] M. Wirsing. Algebraic specifications. In *Handbook of Theoretical Computer Science, Vol. B: Formal Models and Semantics*, pages 675–788. Elsevier, 1990. Edited by J. van Leeuwen.