Proving the correctness of algebraically specified software: Modularity and Observability issues[*]

Gilles Bernot & Michel Bidoit

LIENS – C.N.R.S. U.R.A. 1327
Ecole Normale Supérieure
45, Rue d'Ulm – 75230 PARIS Cedex 05 France
E-mail: [bernot, bidoit] @dmi.ens.fr (Internet), or @FRULM63.BITNET (Earn)

Abstract

We investigate how far modularity and observability issues can contribute to a better understanding of software correctness. We detail the impact of modularity on the semantics of algebraic specifications and we show that, with the stratified loose semantics, software correctness can be established on a module per module basis. We discuss observability issues and we introduce an observational semantics where sort observation is refined by specifying that some operations do not allow observations. Then the stratified loose approach and our observational semantics are integrated together. As a result, we obtain a framework (modular observational specifications) where the definition of software correctness is adequate, i.e. fits with actual software correctness.

1 Introduction

A fundamental aim of formal specifications is to provide a rigorous basis to establish software correctness. Indeed, it is well-known that proving the correctness of some piece of software without a formal reference document makes no sense.[1] Algebraic specifications are widely advocated as being one of the most promising formal specification techniques. However, to be provided with some algebraic specification is not sufficient per se. A precise (and adequate) definition of what does mean the correctness of some piece of software w.r.t. its algebraic specification is mandatory. This crucial prerequisite must be first fulfilled before one can develop relevant verification methods, and try to mechanize them.

Hence the adequacy of the chosen definition of correctness has a great practical impact, and we should therefore define software correctness in conformity with actual needs. In the framework of algebraic specifications, straightforward definitions of correctness turn out to be oversimplified: most programs that must be considered as being correct (from a practical point of view) are rejected. Indeed, when the program behaves correctly, there can still exist some differences between the properties stated by the specification and those verified by the program. Here, to behave correctly means that these differences are not "observable". Consequently, more elaborated definitions of correctness, taking observability into account, should be considered.

[*] **To appear in Proc. of the AMAST Conference, 1991.**

[1] Who would attempt to prove a theorem without providing its statement?

As soon as real-sized systems are involved, both the specification and the software become large and complex. Hence the validation process becomes itself an unmanageable task. At the programming level, this problem is handled by using modular programming languages. At the specification level, algebraic specifications are split into smaller units by means of specification-building primitives. Thus, with respect to software correctness, what is needed is a framework such that the various units of the specification can be related to the various modules of the software, and such that the gobal correctness of the software can be established from the local correctness of each software module w.r.t. its specification module.

Thus, our claim is that modularity and observability issues are fundamental to define a practicable notion of software correctness. In this paper, we will detail various aspects related to modularity, observability, and their interactions with software correctness. We introduce a semantic framework for modular observational algebraic specifications that leads to a more adequate definition of software correctness. This is only a first step towards putting software correctness proofs in practice, but we believe that practicable proof methods can be developed on top of our approach.

We assume that the reader is familiar with algebraic specifications [20, 14] and with the elementary definitions of category theory [25]. An algebraic specification SP is a tuple $(S, \Sigma, \mathcal{A}x)$ where (S, Σ) is a signature and $\mathcal{A}x$ is a finite set of Σ-formulas. We denote by $\mathit{Mod}(\Sigma)$ the category of all Σ-algebras, and by $\mathit{Mod}(\mathit{SP})$ the full sub-category of all Σ-algebras for which $\mathcal{A}x$ is satisfied. We will also use the following technical definition:

> **Definition (Minimal models):**
> Given a specification SP, a model $M \in \mathit{Mod}(\mathit{SP})$ is called *minimal* if, for all $A \in \mathit{Mod}(\mathit{SP})$, if there exists a morphism $\mu : A \longrightarrow M$, then there exists a **unique** morphism $\nu : M \longrightarrow A$.
> Note that if $\mathit{Mod}(\mathit{SP})$ has an initial model, then it is the unique minimal model (up to isomorphism).

2 Modularity and software correctness

In this section we shall focus on the links that can (should) be established between a modular specification and the corresponding software, implemented using a modular programming language (such as e.g. Ada, Clu or Standard ML). The problem considered is to define an algebraic semantic framework such that the various pieces of the specification can be related to the various modules of the implementation and such that the global correctness of the implementation can be established from the local correctness of each software module w.r.t. its specification module.

To better understand why and how far both the modularity of the specification and the modularity of the software interact together as well as the need for a new approach to the semantics of algebraic specifications, we shall first briefly recall the main underlying paradigm of the loose approach.

A specification is supposed to describe a future or existing system in such a way that the properties of the system (**what** the system does) are expressed, and the implementation details (**how** it is done) are omitted. Thus a specification language aims at describing **classes** of correct (w.r.t. the intended purposes) implementations (realizations). In contrast a programming language aims at describing **specific** implementations (realizations). In a loose framework, the semantics of some specification SP is a class $\mathcal{M}$ of (non-isomorphic) algebras. Given some implementation (program) P, its correctness w.r.t. the specification SP can then be established by relating the program P with one of the algebras of the class $\mathcal{M}$. Roughly speaking, the program P will be correct w.r.t. the specification SP if and only if the algebra defined by P belongs to the class $\mathcal{M}$.[2]

Let us now reexamine the above picture in a modular setting. At one hand we have a **modular** specification SP made of some specification modules ΔSP_1, ΔSP_2, ... related to each other by some specification-building primitives. On the other hand we have a **modular** software made of some program modules ΔP_1, ΔP_2, ... Assume moreover that the software structure reflects the specification structure. The problem we have to solve is the following one:

1. To define a notion of correctness such that "the program module ΔP_i is correct w.r.t. the specification module ΔSP_i" is given a precise meaning.

2. To ensure that the local correctness of each program module w.r.t. its specification module implies the global correctness of the whole software w.r.t. the whole specification.

3. To carefully study how some basic requirements about the modular development of modular software, as well as their reusability, interact with the design of the semantics of modular specifications.

It turns out that the main difficulties raised by this goal are twofold:

1. Providing a (loose) semantics to specification **modules** is not so easy, since from a mathematical point of view (heterogeneous) algebras do not have a modular structure.

2. If our intuition and needs about modular software development and the reuse of software modules can be easily figured out, this turns out to be a more difficult task at the level of algebraic semantics.

In the following section we shall try to provide some insight into the solution we propose and into the main ideas underlying what we call the "*stratified loose semantics*".

3 The stratified loose approach

For sake of simplicity, we shall focus on the most commonly used specification-building primitive, namely the enrichment one. Moreover, we shall assume that the modular specification SP_2 we consider is made of one specification module ΔSP that enrich only one modular specification SP_1.

According to the loose approach, the semantics of the specification SP_1 will be defined as some class $\mathcal{M}_1$ of Σ_1-algebras (where Σ_1 denotes the signature associated to SP_1). Similar notations hold for SP_2. Since we assume that SP_2 is defined as an enrichment of SP_1 by the specification module ΔSP, we have $\Sigma_2 = \Sigma_1 \oplus \Delta\Sigma$, hence $\Sigma_1 \subseteq \Sigma_2$. Let $\mathcal{U}$ denote the usual

[2] As we will see in Section 4, this is an oversimplified picture. However, in the sequel we shall adopt this oversimplified understanding of software correctness, since it will be sufficient to study the impact of modularity. Note also that our picture does not preclude more refined views about implementations, such as the abstract implementation of one specification by another (more concrete) one [4, 13], or the stepwise refinement and transformation of a specification into a piece of software [2]. This indeed is the reason why whe shall speak of "realizations" instead of "implementations".

forgetful functor from Σ_2-algebras to Σ_1-algebras.

With the help of this simple context, our intuition and needs w.r.t. the modular development of modular software can be summarized as follows [10]:

1. If some piece of software fulfills (i.e. is a correct realization of) the "large" specification SP_2, then it must be reusable for simpler purposes, i.e. it must also provide a correct realization of the sub-specification SP_1.

2. **Any** piece of software that fulfills (i.e. that is a correct realization of) the sub-specification SP_1 should be reusable as the basis of some correct realization of the larger specification SP_2. In other words, it should be possible to implement the sub-specification SP_1 without taking care of the (future or existing) enrichments of this specification (e.g. by the specification module ΔSP).

3. It should be possible to implement the specification module ΔSP without knowing which specific realization of the sub-specification SP_1 has been (or will be) chosen. Thus, the various specification modules should be implementable **independently** of each other, may be simultaneously by separate programmer teams. Moreover, exchanging some correct realization (say P_1) of the specification SP_1 with another correct one (say P_1') should still produce a correct realization of the whole specification SP_2, without modification of the realization ΔP of the specification module ΔSP.

The first two requirements can be easily achieved by embedding some appropriate *hierarchical constraints* into the semantics of the enrichment specification-building primitive. Roughly speaking, it is sufficient to require the following property:

> *Either $\mathcal{M}_2 = \emptyset$ (in that case the specification module ΔSP will be said to be hierarchically inconsistent) or $\mathcal{U}(\mathcal{M}_2) = \mathcal{M}_1$.*

The third requirement, however, cannot be achieved without providing a suitable (loose) semantics to **specification modules**. There is no way to take this requirement into account by only looking at the semantics of specifications. However, in an initial approach to algebraic semantics (cf. e.g. [14, 12]), an initial semantics can be provided for the specification module ΔSP by considering the free synthesis functor $\mathcal{F}_\Delta$ (left adjoint to the forgetful functor $\mathcal{U}$). In our case, nothing ensures that this free synthesis functor $\mathcal{F}_\Delta$ exists, since we have made no assumption about the axioms of the specification. Moreover, we are looking for a loose semantics of specification modules, in order to reflect **all** correct implementation choices of these modules. The following definition provides the solution we are looking for by embedding the ideas of the initial approach into the loose one:

Definition (Stratified loose semantics):
Given a modular specification SP_2 defined as the enrichment of some modular specification SP_1 by a specification module ΔSP, the semantics of the specification module ΔSP and of the modular specification SP_2 are defined as follows:

Basic case:
If the sub-specification SP_1 is empty (hence the specification SP_2 is reduced to the specification module ΔSP), then:

- The semantics of the specification SP_2 is by definition the class of all minimal models of $Mod(SP_2)$, if any; if $Mod(SP_2)$ has no minimal model, then SP_2 is said to be *inconsistent*.

- The semantics of the specification module ΔSP is defined as being the class of all functors $\mathcal{F}$ from the category $\mathbf{1}$ to $Mod(SP_2)$, which map the object of $\mathbf{1}$ to a minimal model of $Mod(SP_2)$.[3]

General case:

Let us denote by $\mathcal{M}_1$ the class of models associated to the modular specification SP_1, according to the current definition.

- The semantics of the specification module ΔSP is defined as being the class $\mathcal{F}_1^2$ of all the mappings $\mathcal{F}$ such that:
  1. $\mathcal{F}$ is a (**total**) functor from $\mathcal{M}_1$ to $Mod(SP_2)$.
  2. $\mathcal{F}$ is a right inverse of the forgetful functor $\mathcal{U}$, i.e.: $\forall M_1 \in \mathcal{M}_1 : \mathcal{U}(\mathcal{F}(M_1)) = M_1$.

  If the class $\mathcal{F}_1^2$ is empty, then the enrichment is said to be *hierarchically inconsistent.*
- The semantics of the whole specification SP_2 is defined as being the class of all the models image by the functors $\mathcal{F}$ of the models of $\mathcal{M}_1$:
  $$\mathcal{M}_2 = \bigcup_{\mathcal{F} \in \mathcal{F}_1^2} \mathcal{F}(\mathcal{M}_1).$$

The class $\mathcal{M}_2$ of the models of the specification SP_2 is said to be **stratified** by the functors $\mathcal{F}$.

Some comments are necessary to better understand the previous definition:

- Our semantics is a true loose semantics, since it associates a class of (non-isomorphic) functors (resp. algebras) to a given specification module (resp. to a given specification). However, our semantics can also be considered as a generalization of the initial approach: if we restrict to positive conditional equations, then the free synthesis functor from $Mod(SP_1)$ to $Mod(SP_2)$ exists; under suitable additional assumptions, this functor is just one specific functor in the class $\mathcal{F}_1^2$.

- It is important to note that with our loose stratified semantics, the *hierarchical constraints* mentioned above are satisfied. More precisely, as soon as the specification module ΔSP is hierarchically consistent, then we have $\mathcal{U}(\mathcal{M}_2) = \mathcal{M}_1$. As a consequence, both the so-called "*no junk*" and "*no confusion*" properties are guaranteed. In other words, we know that the "old" carrier sets (i.e. the carrier sets of sorts defined in SP_1) will contain no "new" value, and that "old" values who may be distinct before (in at least one model of SP_1) should not be forced to be equal by the new specification module ΔSP.

- We have chosen a pseudo initial semantics (minimal models) for the basic case (a modular specification reduced to one specification module) in order to exclude trivial models (a well-known problematic feature of loose semantics). This remark does not only apply for basic specifications, but for all modular specifications in general, since this "minimal" semantics for the basic case, combined with the hierarchical constraints induced by the stratified loose semantics for the general case, will exclude trivial carrier sets for all sorts.[4]

[3] As usual, the category $\mathbf{1}$ denotes the category containing only one object, which can be interpreted as a Σ_1-algebra for an empty signature Σ_1.

[4] More precisely, if we consider a sort s and if we assume that there exists some operation (or some composition of operations) which has s in its domain and a sort s' defined in a basic specification module as its codomain, then the "minimal" semantics of the basic specification module will prevent from undesired confusion of values in the carrier set of s ...

Moreover, such a "minimal" semantics for basic specification modules turns out to be quite adequate to specify enumerated sets (such as e.g. booleans, characters, etc.).

- So far, we have stressed that the independent implementability of each specification module is a crucial aspect of modularity. Now, we would like to stress another equally important aspect of modularity, namely the specification style point of view. Indeed, when writing some specification module, a natural implicit assumption made by the specifier is that the semantics of the imported sub-specifications is preserved (i.e. this semantics is established once for all). By the way, this is exactly what is guaranteed by our stratified loose semantics: as explained aboved, the hierarchical constraints associated to our semantics of modularity automatically restrict the class of models of the specification SP_2 to the models that preserve the enriched sub-specifications. This contrasts with a more conventional approach, where the specifier should explicitly design the axioms of the specification unit in order to guarantee the so-called no junk and no confusion properties, i.e. to guarantee the persistency of the enrichment (and this often results in unnecessary over-specification). From our point of view, there is therefore a fundamental distinction between what we call **structured specifications**, for which the no junk and no confusion properties are **explicitly** ensured by appropriate axioms, and **modular specifications**, for which similar properties are **implicitly** ensured by an appropriate semantics. Furthermore, it is clear that the hierarchical structure of a modular specification has a deep impact on its modular semantics, while this is not the case for (persistent) structured specifications, whose semantics is not altered by flattening.

- The extension of the definition above to the case where the specification module ΔSP enriches more than one specification as well as its extension to other specification-building primitives (such as e.g. parameterization) do not raise difficult problems and is described in [9].

- It is also important to note that our definition is independent of the underlying institution [19]. Thus our stratified loose approach can be used to define the semantics of any modular algebraic specification language [33]. Moreover, our stratified loose approach can even be used in a more general framework than institutions, for instance in a framework where the *Satisfaction Condition* [19] does not hold. This is obvious since, as far as the stratified loose semantics is concerned, the existence of forgetful functors (from Σ_2-algebras to Σ_1-algebras, with $\Sigma_1 \subseteq \Sigma_2$) is the only requirement. Indeed, this very broad scope of the stratified loose approach will be clearly demonstrated in Section 6.

We must now point out how far our stratified loose semantics solves the problem stated in the previous section. A program module ΔP will be said to be correct w.r.t. some specification module ΔSP if and only if ΔP "defines" a functor belonging to the semantics of the specification module. From our definition, it is then obvious that the "composition" of correct software modules (i.e. the software obtained by linking together these software modules) is always a correct realization of the whole specification. Thus, the main significance of the stratified loose framework outlined in this section is that it is possible to specify and develop software in a modular way, and that the correctness of the implementation should only be established on a module per module basis. A formal theory of software reusability, built on top of our stratified loose semantics, is described in [17].

Note that the definition above is given in a very general way: we have considered **all** algebras, finitely generated or not. It is obviously very easy to refine our definition of the stratified loose semantics in order to consider finitely generated algebras only. Indeed, we prefer to introduce a more powerful constraint, namely the restriction to algebras finitely generated with respect to a distinguished subset of the signature called the set of *generators*.[5] Such a constraint will guarantee that for any model, all values will be denotable as some composition of these generators. From a theoretical point of view, an important consequence of this constraint is that *structural induction restricted to the generators* is a correct proof principle. This constraint has many practical consequences too, since reasoning by means of generators helps writing the axioms in a structured way [7]. Moreover, it can be used to avoid to overspecify some operations, as demonstrated in the following two examples:

Specifying a *remove* operation on sets :
To specify a *remove* operation on sets, the following axioms are sufficient:

$$x \in remove(x,S) = false$$
$$x{\neq}y \implies y \in remove(x,S) = y \in S$$

Considering only finitely generated models w.r.t. the generators will ensure that, for any set S, $remove(S)$ is actually a set, reachable from the empty set by some successive insertions, and not a “junk” set.

Specifying the Euclidean division :
Similarly, to specify the division of natural numbers, the following axioms are sufficient:

$$m{\neq}0 \implies 0 \leq [n - ((n\ div\ m) * m)] = true$$
$$m{\neq}0 \implies m \leq [n - ((n\ div\ m) * m)] = false$$

These axioms characterize (n *div* m) among all natural numbers *finitely generated w.r.t.* 0 *and succ* (Euclid). However, without the constraint, there are models (e.g. the initial model) where (n *div* m) is not reached by some $succ^i(0)$; it is only an unreachable value such that the (unreachable) remainder $[n - ((n\ div\ m) * m)]$ returns the specified boolean values when compared with 0 and m .

In Pluss [9], the distinguished subset of generators is specified apart from the other operations of the signature and is introduced by the keyword **generated by**.

A crucial issue is obviously to know when some given specification module is *hierarchically consistent*. From a general point of view, it is well-known that this is an undecidable problem. However, we would like to point out that, in our stratified loose framework, there are two distinct grounds for hierarchical inconsistency:

- As usual, hierarchical inconsistency may result from the axioms introduced by the specification module.
- Moreover, adding **“new” observations on “old” sorts** will in general result in an inconsistent specification module ΔSP. This is due to the fact that, with these “new” observations, it may be possible to distinguish “old” values (i.e. to prevent them from being equal), while these “old” values could have been equal in some model M_1 of SP_1. Hence there is no **total** mapping from $\mathcal{M}_1$ to $Mod(SP_2)$, since this model M_1 of SP_1

[5] We do not detail here the refined version of the stratified loose semantics according to this additional constraint, since the modifications to be introduced are rather obvious [9].

cannot be extended to a model of SP_2. A typical example of such a situation is when a "new observing operation" on an "old" sort is defined in ΔSP: for instance, if we assume that SP_1 specifies natural numbers (with $\mathcal{M}_1 \supseteq \{\mathbf{N},\ \mathbf{Z/nZ}\}$), specifying a "$\leq$" operation in ΔSP will result in an hierarchically inconsistent specification module. Thus, these "observing operations" should rather have been defined in the appropriate specification module, i.e. in the specification module where the "observed" sort is defined.

Note that the latter ground for hierarchical inconsistency should not be understood as a restrictive side-effect of the stratified loose semantics, but rather as a fruitful guide in structuring large specifications into specification modules. More precisely, a specification module should be considered as a unit of specification where a sort of interest, its generators, its observers, and other appropriate operations are simultaneously defined.

As a consequence of the "hierarchical constraints" required by modularity, it is necessary to state a careful distinction between *implementable* and *not yet implementable* specification modules:

- *Implementable specification modules* will have a semantics defined accordingly to the stratified loose framework, in order to allow for a **modular** software development and verification process.

- *Not yet implementable specification modules* will have a more flexible semantics, in order to allow for a specification development process by stepwise refinements [8].

Such a distinction is introduced in the **Pluss** algebraic specification language [9, 11], the semantics of which is defined following the stratified loose approach. Note that this distinction contrasts with all other specification languages developed following either the initial or the loose approach, such as ASL [37, 1], OBJ2 [15] and LARCH [21], where there is only a distinction between various enrichment primitives.

4 Observability and software correctness

The paradigm used in Section 2 to introduce the stratified loose approach was obviously an oversimplified understanding of software correctness. Indeed, if software correctness (w.r.t. its formal specification) is defined in such a way, then most realizations that we would like to consider as being correct (from a practical point of view) turn out to be incorrect ones. This is illustrated by the *SET* specification given in Fig. 1.

If we consider a standard realization of *SET* by e.g. lists, we do not obtain a correct realization: this is due to the axioms expressing the commutativity of the insertion operation, which do not hold for lists. However, if we notice that indeed we are only interested in the result of some computations (e.g. membership), then it is clear that our realization of *SET* by lists "behaves" correctly. Thus, an intuitively correct realization of an algebraic specification *SP* may correspond to an algebra which is **not** in *Mod*(*SP*). This leads to a refined understanding of software correctness: a program P should be considered as being correct w.r.t. its specification *SP* if and only if the algebra defined by P is an "observationally correct realization" of *SP*. In other words, the differences between the specification and the software should not be "observable", w.r.t. some appropriate notion of "observability".

```
spec : SET ;
    use : NAT, BOOL ;
  sort : Set ;
  generated by :
    ∅ : ⟶ Set ;
    ins: Nat Set ⟶ Set ;
  operations :
    _ ∈ _ : Nat Set ⟶ Bool ;
    del : Nat Set ⟶ Set ;
  axioms :
    ins(x, ins(x, S)) = ins(x, S) ;
    ins(x, ins(y, S)) = ins(y, ins(x, S)) ;
    del(x, ∅) = ∅ ;
    del(x, ins(x, S)) = del(x, S) ;
    x ≠ y ⟹ del(x, ins(y, S)) = ins(y, del(x, S)) ;
    x ∈ ∅ = false ;
    x ∈ ins(x, S) = true ;
    x ≠ y ⟹ x ∈ ins(y, S) = x ∈ S ;
    where :   S : Set ;   x, y : Nat ;
end SET .
```

Figure 1: A specification of sets of natural numbers

The problem is now to specify the "observations" to be associated to some specification, and to define the semantics of such "observations" in order to obtain a framework that will capture the essence of software correctness. Up to now, various notions of observability have been introduced, involving observation techniques based on sorts [18], [36], [24], [16], [30], [26], [34], [29], [28], on operations [35], on terms [32], [23] or on formulas [31], [32]. Assuming that we have chosen some observation technique, we can specify, using this technique, that some parts of an algebraic specification are observable. An observational specification is thus obtained by adding a specification of the objects to be observed to a usual algebraic specification. The next step is to define the semantics of these observational specifications, in such a way that our paradigm "the class of the models of some specification represents all its acceptable realizations" is correctly reflected. As explained above, some correct software could correspond to an algebra which does not satisfy all the axioms of the specification, provided that the differences between the properties of the algebra and the properties required by the specification are not observable.

There are mainly two possible ways to define the semantics of observational specifications. We can extend the class of the models of the specification *SP* by including some additional algebras which are "behaviourally equivalent" (w.r.t. the specified observations) to a model of *Mod*(*SP*) (*extension by behavioural equivalence*, see [31], [32], [23]). In the sequel such an approach will be referred to as **behavioural semantics**. We can also directly relax the satisfaction relation, hence redefine *Mod*(*SP*) (*extension by relaxing the satisfaction relation*, see [30], [29], [35]). We will call these approaches **observational semantics**.

For a comparative study of these various ways of defining the semantics of observational specifications, and of the relative expressive power of the various observation techniques mentioned above, see [5]. In the sequel we will provide a short insight into the behavioural approach, and we will point out some of its limitations. In the next section we will describe a semantics based on the observational approach.

To define a behavioural semantics we first need to define an appropriate equivalence relation $\equiv_{Obs}$ on the class $Mod(\Sigma)$ of all Σ-algebras, also called behavioural equivalence of algebras w.r.t. the specified observations *Obs* [31, 32]. The definition of $\equiv_{Obs}$ depends on the observational technique in use (i.e. whether we observe sorts, operations, terms or formulas [5]). Assuming that we observe a set of formulas Φ (which is the most general case), the behavioural equivalence $\equiv_\Phi$ and the associated behavioural semantics are defined as follows:

> **Definition (Behavioural semantics) [32]:**
> Given a set of observed formulas Φ, the behavioural equivalence w.r.t. Φ, written $\equiv_\Phi$, is an equivalence relation on $Mod(\Sigma)$ defined by:
>
> $$A \equiv_\Phi B \quad \textit{if and only if} \quad \forall \varphi \in \Phi \quad A \models \varphi \quad \Leftrightarrow \quad B \models \varphi$$
>
> In other words, two Σ-algebras A and B are behaviourally equivalent w.r.t. a set of observable formulas Φ, if and only if A and B satisfy the same observable formulas. The class of the behavioural models of some specification *SP* (with observed formulas Φ), written $Beh(SP, \Phi)$, is defined by:
>
> $$Beh(SP, \Phi) = \{B \in Mod(\Sigma) \;\; | \;\; \exists\ A \in Mod(SP)\ s.t.\ B \equiv_\Phi A\}$$

Now we would like to point out some limitations intrinsic to behavioural semantics. It turns out that in some cases, behavioural semantics is not powerful enough to fully capture our requirements w.r.t. software correctness: in these cases, we know of some realization that we would like to consider as being correct, but unfortunately this realization cannot be shown to be behaviourally equivalent to any of the (usual) models of the specification at hand. A typical example of such cases arises when $Mod(SP)$ is empty (i.e. when the specification is inconsistent in the usual sense). For instance, let us consider a variant of our *SET* specification as described in Fig. 2.

What we really need for this example is to observe the following set of terms:

$$W \;=\; \{x \in S\} \cup \{t \in T_{LIST\text{-}signature}(X) \mid t \textit{ is of sort Nat or Bool }\}$$

In other words, we observe membership and some *LIST* terms but we do not observe those *LIST* terms where *enum* occurs.[6]

Obviously, the specification *SET-WITH-ENUM* is inconsistent (i.e. $Mod(SP) = \emptyset$). Consequently its class of behavioural models is empty as well, whatever the observations specified and the behavioural equivalence used. Nevertheless, a realization which represents sets by non redundant lists, *ins* being realized by *cons* (when the element to be inserted is not already in the list) and *enum* being a coercion, should clearly be considered as a correct one.

The point here is that in a behavioural approach, the existence of behavioural models depends on the existence of usual models. Indeed, behavioural semantics still rely on the usual

[6] Note that for this example we observe terms and not formulas. However, as shown in [5], behavioural equivalence can be defined in a similar way as above. Moreover, whatever the definition of $\equiv_{Obs}$ is, our counter-example remains.

```
spec : SET-WITH-ENUM ;
      use : LIST, NAT, BOOL ;
   sort : Set ;
   generated by :
      ∅ : ⟶ Set ;
      ins: Nat Set ⟶ Set ;
   operations :
      _ ∈ _ : Nat Set ⟶ Bool ;
      del : Nat Set ⟶ Set ;
      enum : Set ⟶ List ;
   axioms :
      ins(x, ins(x, S)) = ins(x, S) ;
      ins(x, ins(y, S)) = ins(y, ins(x, S)) ;
      del(x, ∅) = ∅ ;
      del(x, ins(x, S)) = del(x, S) ;
      x ≠ y ⟹ del(x, ins(y, S)) = ins(y, del(x, S)) ;
      x ∈ ∅ = false ;
      x ∈ ins(x, S) = true ;
      x ≠ y ⟹ x ∈ ins(y, S) = x ∈ S ;
      enum(∅) = nil ;
      x ∈ S = true ⟹ enum(ins(x, S)) = enum(S) ;
      x ∈ S = false ⟹ enum(ins(x, S)) = cons(x, enum(S)) ;
      where :   S : Set ;   x, y : Nat ;
end SET-WITH-ENUM .
```

Figure 2: A variant of the specification of sets of natural numbers

satisfaction relation, hence behavioural consistency coincides with standard consistency. This is the reason why we shall develop in the next section an approach based on observational semantics, i.e. an approach where the satisfaction relation is redefined accordingly to the specified observations.

5 Observational specifications

In this section we develop a new framework for observational specifications, the semantics of which is based on a redefinition of the satisfaction relation. We will only consider flat or structured observational specifications, modular ones being dealt with in the next section.

As explained in the previous section, we want to reflect the following idea: some data structures are observable with respect to some observable sorts (e.g. lists are observable w.r.t. their elements via terms such as $car(L)$ or $car(cdr(L))$ etc.), but we need also to prevent from observing the results of some specific operations (e.g. if a list is obtained by enumeration of a set, $enum(S)$, it must not be observed; in particular, $car(enum(S))$, which denotes a value of an observable sort, must nevertheless be non observable). This leads to the following idea: given a specification *SP*, one defines the set of *observable sorts* S_{Obs}, and in addition one defines the set of "operations allowing observations" which is a subset Σ_{Obs} of the signature of *SP* (e.g. Σ_{Obs}

can contain all the operations except *enum*).

5.1 Definitions

Let us first define the syntax of (flat) observational specifications.

Definition (Observational specifications):

- An *observation Obs* over a signature (S, Σ) is a couple (S_{Obs}, Σ_{Obs}) such that $S_{Obs} \subseteq S$ and $\Sigma_{Obs} \subseteq \Sigma$.
- An *observational signature* is a couple (Σ, Obs) such that *Obs* is an observation over Σ.
- An *axiom* over a signature Σ is a sentence whose atoms are equalities (between two Σ-terms of the same sort, with variables) and whose connectives belong to $\{ \neg, \wedge, \vee, \Rightarrow \}$. Every variable is implicitly universally quantified.
- An *observational specification* is a couple *SP–Obs* $= (SP, Obs)$ such that $SP = (S, \Sigma, \mathcal{A}x)$ is a classical specification (i.e. $\mathcal{A}x$ is a set of axioms over Σ) and *Obs* is an observation over the signature Σ.
- If $\mathcal{A}x$ only contains equalities then *SP–Obs* is called *equational*. If $\mathcal{A}x$ only contains axioms of the form $[(u_1 = v_1) \wedge \ldots \wedge (u_n = v_n) \implies (u = v)]$ then *SP–Obs* is called *positive conditional*.

Example:
We have seen that, for the specification *SET-WITH-ENUM* given in Fig. 2, we need to observe only the terms of sort *Bool* or *Nat* where the operation *enum* does not occur. Thus, it is sufficient to declare $S_{Obs} = \{Bool, Nat\}$ and $\Sigma_{Obs} = \Sigma - \{enum\}$ in order to obtain the required set of observable terms W already mentioned in Section 4.

As usual, the notion of *observable contexts* is crucial for observability [24, 30, 29, 23, 22]:

Definition (Observable contexts):

- In general a *context* over a signature Σ is a Σ-term C with exactly one occurrence of one variable.
- Given a context C, its *arity* is $(s' \to s)$, where s' is the sort of the variable occurring in C and s is the sort of (the term) C. s is also called the (target) sort of C.
- Let (Σ, Obs) be an observational signature; the associated set of *observable contexts* is the set $\mathcal{C}_{Obs}$ which contains all the contexts over the signature Σ_{Obs} whose target sort belongs to S_{Obs}.
- For each observable sort $s \in S_{Obs}$, the context reduced to a variable of sort s is called “the empty context” (of sort s).

Let us now define the semantics of (flat) observational specifications.

Definition (Observational semantics):
Let *SP–Obs* be an observational specification and Σ be its signature. Let M be a Σ-algebra and let ax be an axiom of *SP–Obs*.

- Two elements a and b of M are *observationally equal* with respect to *Obs* if and only if they have the same sort s and for all contexts $C \in \mathcal{C}_{Obs}$ of arity $s \rightarrow s'$, $C(a) = C(b)$ in M (according to the usual equality of set theory). In particular observational equality on observable sorts coincides with the set-theoretic equality;[7] for the non observable sorts, the observational equality contains the set-theoretic equality, but there are also distinct values which are observationally equal.
- The algebra M *satisfies ax* with respect to *Obs* means that for all substitutions $\sigma : T_\Sigma(X) \rightarrow M$, $\sigma(ax)$ holds in M according to the observational equality (defined above) and the truth tables of the connectives.
- The algebra M *satisfies SP–Obs* means that it satisfies all the axioms of *SP–Obs* with respect to *Obs*.
- The satisfaction of observational equalities is denoted by "$\models_{Obs}$" and we write "$M \models_{Obs} (a = b)$", "$M \models_{Obs}$ *SP–Obs*"...

Example:
It is not difficult to show that the realization of *SET-WITH-ENUM* by non redundant lists described in the previous section satisfies the observational specification given above. For instance:

- When $x \notin S$, $enum(insert(x, S)) = cons(x, enum(S))$ is satisfied because they are equal (with respect to the set-theoretic equality) in our model; thus, they are a fortiori observationally equal.
- $insert(x, insert(y, S)) = insert(y, insert(x, S))$ is observationally satisfied (even when these two list realizations of sets are not equal with respect to the set-theoretic equality) because all contexts involving *enum* do not belong to $\mathcal{C}_{Obs}$; here, all the observable contexts C in $\mathcal{C}_{Obs}$ have $Set \rightarrow Bool$ as arity and the top symbol of C is necessarily "$\in$".

Notation:
Given an observational specification *SP–Obs*, *Mod*(*SP–Obs*) is the full sub-category of *Mod*(Σ) whose objects are the Σ-algebras satisfying *SP–Obs*.

The following results are trivial:

Fact-1:
Given an observational signature (Σ, Obs), Σ-morphisms preserve observational equalities: for all $\mu : M \rightarrow M'$, if $M \models_{Obs} (a = b)$ then $M' \models_{Obs} (\mu(a) = \mu(b))$.

Fact-2:
Mod(*SP*) is equal to *Mod*(*SP–Obs*) when $S_{Obs} = S$ (due to the empty contexts).

Fact-3:
If *SP–Obs* is equational then the usual category *Mod*(*SP*) is a full sub-category of *Mod*(*SP–Obs*).

Fact-4:
More generally, if *SP* is an equational specification and if $Obs_1 \subseteq Obs_2$ then *Mod*(*SP–Obs*$_2$) is a full sub-category of *Mod*(*SP–Obs*$_1$).

[7] because $\mathcal{C}_{Obs}$ always contains the empty contexts on observable sorts.

Notice that, from Fact-2, Fact-3 is a particular case of Fact-4. Moreover, the inclusions stated in Fact-3 (hence in Fact-4) are often strict: there is often a model M with two elements $a{\neq}b$ such that $M{\models}_{Obs}(a = b)$.

Fact-5:
Fact-3, hence Fact-4, cannot be extended to non equational specifications. For example let M be an algebra such that $a{\neq}b$ and $c{\neq}d$ with $M{\models}_{Obs}(a = b)$ and $M{\not\models}_{Obs}(c = d)$. Then, M satisfies $[(a = b) \;\Rightarrow\; (c = d)]$ in the classical sense because the precondition is false, but it does not satisfy this axiom with respect to *Obs*.

When flattened, our specification of *SET-WITH-ENUM* is an example where *Mod*(*SP*) only contains algebras with a trivial *Nat* carrier (a singleton), but *Mod*(*SP–Obs*) contains, among others, algebras where the *Nat* part is isomorphic to **N**.

5.2 Initiality results

As usual, initiality results can only be easily obtained for equational, or positive conditional, specifications [38].

Theorem (Least congruence):
Let *SP–Obs* be a **positive conditional** observational specification and M be a Σ-algebra. There exists a least congruence $\equiv$ on M such that the quotient algebra $M/_{\equiv}$ satisfies *SP–Obs*.

Sketch of the proof: Let F be the family of all the congruences such that $M/_{\equiv}$ satisfies *SP–Obs*. It is not empty because the trivial congruence τ defined by $(a \;\tau\; b) \;\Leftrightarrow\; (a \textit{ and } b \textit{ have the same sort})$ belongs to F. Let $\equiv$ be the intersection of all the congruences in F; if $\equiv$ still belongs to F then the theorem is proved. Consequently, we simply have to prove that $M/_{\equiv}$ satisfies (observationally) each axiom of *SP*. This is not difficult, by applying the definitions.

The following corollary is simply obtained from the previous theorem with $M = T_\Sigma$ (as in [20]):

Corollary (Initial object):
The category *Mod*(*SP–Obs*) has an initial object $I = (T_\Sigma/_{\equiv})$.

It may seem surprising that, regardless of several other works on observability [24, 27, 28], we care about initial objects while they care about terminal ones. Indeed we believe that any "collapse between values" reflects some **implementation choice** (each implementation choice being intuitively reflected by some equation which induces new collapses). From this viewpoint, "considering the least congruence" means "considering only the necessary implementation choices"; thus, the initial algebra can be considered as the most general realization. More generally, when the initial algebra does not exist, minimal models can be considered as realizations with "minimal implementation choices". Moreover, *Mod*(*SP–Obs*) has always a terminal object which is the trivial algebra. This algebra has clearly no interest. This is due to the fact that, for the moment, our specifications are **flat**. Terminal algebras get interest only when some enriched specifications are "protected". We shall consider such hierarchical constraints when observational semantics and the stratified loose approach will be integrated together.

5.3 Structured observational specifications

The following proposition generalizes Fact-4 above.

Proposition:
Let *SP–Obs*$_1$ and *SP–Obs*$_2$ be two observational specifications such that *SP–Obs*$_1 \subseteq$ *SP–Obs*$_2$. If *SP–Obs*$_1$ is equational then the forgetful functor $\mathcal{U}$ from $Mod(\Sigma_2)$ to $Mod(\Sigma_1)$ has the following property:
For all Σ_2-algebras M satisfying *SP–Obs*$_2$, the Σ_1-algebra $\mathcal{U}(M)$ satisfies *SP–Obs*$_1$.
Then $\mathcal{U}$ also denotes the forgetful functor from *Mod*(*SP–Obs*$_2$) to *Mod*(*SP–Obs*$_1$).

Proof: Results from $\mathcal{C}_{Obs-1} \subseteq \mathcal{C}_{Obs-2}$.

This proposition can be extended to positive conditional observational specifications whose axioms only contains equalities of **observable sorts** (in S_{Obs}) **in the preconditions**. Such an extension is similar to some sufficient conditions used in [22, 6] for proof methods with observability.

Note that, from Fact-5 above, this proposition cannot in general be extended to a non equational specification *SP–Obs*$_1$. Moreover, for the same reason, observational specifications do not define an institution [19] because the *Satisfaction Condition* is not guaranteed in our framework. Anyway, it seems clear that this counter-fact is intrinsic to the observability question: there is no reasonable syntactical constraint which ensures that an enrichment does not add new observations of old values. Consequently some axioms which were satisfied by the models of a specification can become unsatisfied when new observations are added. This can only be handled through modularity constraints, which cannot be easily reflected within the institution framework (just because of the *Satisfaction Condition*). As explained later on, we shall reflect these constraints owing to the stratified loose semantics.

Provided that the forgetful functor $\mathcal{U}$ exists (from *Mod*(*SP–Obs*$_2$) to *Mod*(*SP–Obs*$_1$)), and that *SP–Obs*$_2$ is positive conditional, $\mathcal{U}$ as a left adjoint, as stated in the following theorem:

Theorem (Free synthesis functor):
Let *SP–Obs*$_1$ and *SP–Obs*$_2$ be two observational specifications such that *SP–Obs*$_1 \subseteq$ *SP–Obs*$_2$. If *SP–Obs*$_1$ is equational and *SP–Obs*$_2$ is positive conditional then the forgetful functor $\mathcal{U}$ admits a left adjoint functor $\mathcal{F}_\Delta$ from *Mod*(*SP–Obs*$_1$) to *Mod*(*SP–Obs*$_2$). In particular, there is a unit of adjunction which provides a canonical Σ_1-morphism from M to $\mathcal{U}(\mathcal{F}_\Delta(M))$ for every algebra M in *Mod*(*SP–Obs*$_1$).

Sketch of the proof: We use the existence of a minimal congruence exactly as for the classical ADJ framework of algebraic specifications with positive conditional axioms. For each $M \in$ *Mod*(*SP–Obs*$_1$) we consider the Σ_2-algebra $T_{\Sigma_2}[M]$. Let $\equiv$ be the least congruence on $T_{\Sigma_2}[M]$ generated by the (observational) axioms of *SP–Obs*$_2$ and the fibers of the canonical morphism from $T_{\Sigma_1}[M]$ to M. The *SP–Obs*$_2$ algebra $T_{\Sigma_2}[M]/_\equiv$ is by definition $\mathcal{F}_\Delta(M)$. It is not difficult (but tedious!) to prove that $\mathcal{F}_\Delta$ is compatible with the morphisms (thus it is a functor) and that there is a natural bijection between $Hom_{SP\text{–}Obs_1}(X,\mathcal{U}(Y))$ and $Hom_{SP\text{–}Obs_2}(\mathcal{F}_\Delta(X),Y)$ for all objects $X \in$ *Mod*(*SP–Obs*$_1$) and $Y \in$ *Mod*(*SP–Obs*$_2$) (which is the definition of adjunction).

As usual, the existence of the unit of adjunction allows to define *hierarchical consistency* within an initial framework [3].

Definition (Hierarchical consistency):
Let *SP–Obs*$_1$ and *SP–Obs*$_2$ be two observational specifications such that *SP–Obs*$_1$ $\subseteq$ *SP–Obs*$_2$, *SP–Obs*$_1$ is equational and *SP–Obs*$_2$ is positive conditional. The observational specification *SP–Obs*$_2$ is *hierarchically consistent* w.r.t. *SP–Obs*$_1$ if and only if the canonical morphism from I_1 to $\mathcal{U}(I_2)$ is a monomorphism (i.e. is injective in our framework), where I_1 (resp. I_2) denotes the initial algebra of *Mod*(*SP–Obs*$_1$) (resp. *Mod*(*SP–Obs*$_2$)).

Remember that left adjoint functors preserve initial models, thus $I_2 = \mathcal{F}_\Delta(I_1)$.

Unfortunately, a similar definition of *sufficient completeness* (via the surjectivity of the canonical morphism from I_1 to $\mathcal{U}(I_2)$) is not adequate. For example, when considering our *SET-WITH-ENUM* enrichment, this canonical morphism is not an epimorphism: in the initial object I_2, the terms $enum(S)$ create new list values which are only **observationally** equal to old list values. Thus, the following definition could be better:

Definition (Sufficient completeness):
Let *SP–Obs*$_1$ and *SP–Obs*$_2$ be two observational specifications such that *SP–Obs*$_1$ $\subseteq$ *SP–Obs*$_2$, *SP–Obs*$_1$ is equational and *SP–Obs*$_2$ is positive conditional. The observational specification *SP–Obs*$_2$ is *sufficiently complete* w.r.t. *SP–Obs*$_1$ if and only if the canonical morphism μ from I_1 to $\mathcal{U}(I_2)$ has the following property:
For all values $v \in \mathcal{U}(I_2)$, there exists a value $u \in I_1$ such that $\mathcal{U}(I_2) \models_{Obs} (v = \mu(u))$.

This allows us to define *persistency*:

Definition (Persistency):
Let *SP–Obs*$_1$ and *SP–Obs*$_2$ be two observational specifications such that *SP–Obs*$_1$ $\subseteq$ *SP–Obs*$_2$, *SP–Obs*$_1$ is equational and *SP–Obs*$_2$ is positive conditional. The observational specification *SP–Obs*$_2$ is *persistent* w.r.t. *SP–Obs*$_1$ if and only if it is hierarchically consistent and sufficiently complete.

Of course, such an initial approach is rather restrictive. We must more or less restrict ourselves to equational specifications in order to exploit the results stated in this section. However, almost all the classical results build on the top of the ADJ group approach are then usable. In particular, if a specification has been written following the “fair presentation” method then it is sufficiently complete, and if there are no explicit equations between generators then it is persistent [7].

Nevertheless, we believe that our definition of sufficient completeness is not fully satisfactory because I_2 does not protect the predefined data structure reflected by I_1. Indeed, our realization of sets by non redundant lists already described is a suitable model, while I_2 is not a suitable model. This means that the initial approach is not fully adequate: hierarchical constraints should be substituted to sufficient completeness. More generally, structured specifications are probably not powerful enough to capture the essence of observability. In other words, we believe that observability issues intrinsically require a modular approach with semantic constraints.

6 Integrating observability and the stratified loose approach together: Modular observational specifications

In this section we show how we can obtain a satisfactory approach to software correctness by embedding observability into the stratified loose approach defined in Section 3.

Remember that when we have defined the stratified loose semantics of modular specifications in Section 3, we have claimed that this definition was (more than) institution independent. We will benefit here from this property, since considering modular observational specifications (with the observational semantics defined in the previous section) instead of standard modular specifications directly provides us with the adequate semantics we are looking for. To be more precise, we have explained in the previous section that the observational semantics we have introduced does not lead to an institution of observational specifications. However, since the existence of forgetful functors from Σ_2-algebras to Σ_1-algebras is the only requirement really needed for the stratified loose approach, there is no difficulty to translate the definition of the stratified loose semantics for modular observational specifications.

Thus, combining the stratified loose semantics (for modularity) with the observational semantics defined in Section 5 provides a framework where:

- The global correctness of some software w.r.t. its formal specification can be established on a module per module basis.
- Local correctness is defined in a way flexible enough to cope with "non observable" differences between the properties of the software module and the properties specified by the corresponding specification module.

The crucial point here is that the hierarchical constraints induced by the stratified loose semantics will guarantee us that the composition of correct software modules will always result in a correct software, and that the various modules of the specification can be implemented independently of each other. Hence we do not have to worry about the somewhat problematic features discussed at the end of Section 5. More precisely, the "no junk" and "no confusion" properties inherent to the stratified loose approach (cf. Section 3) are still valid here, and there is no need for the definitions of "sufficient completeness" and "hierarchical consistency" given in the initial approach to structured observational specifications. Moreover, in the previous section we have provided arguments for demonstrating that our observational semantics is powerful (i.e. "flexible") enough, since the counter example discussed at the end of Section 4 was solved in an elegant way by an adequate redefinition of the satisfaction relation.

We believe that the framework developed in this paper provides a firm basis to establish the correctness of some (modular) software w.r.t. its (modular, observational) specification. However, if we really want to prove the correctness of some software, then we need adequate deduction rules and proof techniques. This point is far beyond the scope of this paper, but we would nevertheless discuss some proof related aspects. Remember that in Section 3 we have introduced the restriction to finitely generated models (w.r.t. the operations specified as generators) to guarantee that "induction w.r.t. the generators" is a correct proof principle. An obvious question is whether a similar restriction can be introduced in the framework of observational specifications, and whether we will obtain a similar powerful proof principle.

As a first remark we should note that the restriction to finitely generated models (w.r.t. the generators) is not adequate since such a restriction will be somehow contradictory with the aim of the observational semantics we have developed so far. To illustrate this we will consider the following example:

Example (Stacks implemented by arrays):
Let us consider an observational specification of stacks of natural numbers where

S_{Obs} is the singleton $\{Nat\}$ and Σ_{Obs} is equal to Σ; the generators being obviously *emptystack* and *push* for the sort *Stack*. Of course, we would like to consider a model which implements stacks by means of arrays as an observationally correct model: stack values are couples (a, h) where a is an array and h is the height of the stack; *emptystack* is realized by some initial array $a = init$ and $h = 0$, *push* records its element at range h in a and increments h, *pop* simply decreases h without modifying a, etc.

Let us assume that the initial array *init* uniformly contains 0 for all indices. Then, all stacks values obtained via the generators *emptystack* and *push* satisfy the following property: for all indices $i \geq h$, $a[i] = 0$. But this property is not satisfied for the stack $pop(push(1, emptystack))$ (because $h = 0$ and $a[0] = 1$ for this stack value). Consequently this model is not finitely generated w.r.t. the generators *emptystack* and *push*.

Nevertheless, one should remark that even though $pop(push(1, emptystack))$ is not equal to *emptystack* according to the set-theoretic equality in our model, it is **observationally equal** to *emptystack* .

Thus, it is clear that we must allow values that are not denotable by a composition of generators; but we can still obtain the desired proof principle by requiring for each value to be observationally equal to a value denotable by a composition of generators. This leads to the following definition.

Definition (Observational restriction to generators):
Let *SP–Obs* be a modular observational specification. Let $\Omega \subseteq \Sigma$ be the set of generators declared in *SP–Obs*. A model M of *SP–Obs* is *observationally finitely generated w.r.t.* Ω if and only if for every value m in M there exists an Ω-term t such that $M \models_{Obs} (m = t)$.

As a second remark, we would like to point out that refining the stratified loose semantics with the "observational restriction to generators" constraint has at least two advantages. It simplifies the proof principles and moreover, it has an important consequence on the "specification style": some operations can be specified in a really abstract manner, as demonstrated in the following toy example:

Example (pickout in sets):
Let us consider a specification of sets with a *pickout* operation which is supposed to delete **one of** the elements of a set. We do not want to specify **which** element has to be deleted. This specification module is described in Fig. 3.

For sake of simplicity, let us assume that the elements are the boolean values. The models that we clearly would like to accept are the ones which contain four set values up to observational equality: $\emptyset$, $\{true\}$, $\{false\}$ and $\{true, false\}$. Two possible behaviours of *pickout* are acceptable: $pickout(\{true, false\}) = \{true\}$ or $pickout(\{true, false\}) = \{false\}$.[8] As a matter of fact, we exactly get these models when we consider the semantic constraint of "observational restriction to generators". If this constraint is not required, then we get exotic models such as:

- The set carrier contains five values: $\emptyset$, $\{true\}$, $\{false\}$, $\{true, false\}$ and a strange set $\{true{+}false\}$.

[8] These equalities are only **observational** ones.

```
spec : SET-WITH-PICKOUT ;
    use : ELEM, NAT, BOOL ;
  sort : Set ;
  generated by :
    ∅ : ⟶ Set ;
    ins: Elem Set ⟶ Set ;
  operations :
    _ ∈ _ : Elem Set ⟶ Bool ;
    card : Set ⟶ Nat ;
    pickout : Set ⟶ Set ;
  axioms :
    ins(x, ins(x, S)) = ins(x, S) ;
    ins(x, ins(y, S)) = ins(y, ins(x, S)) ;
    x ∈ ∅ = false ;
    x ∈ ins(x, S) = true ;
    x ≠ y ⟹ x ∈ ins(y, S) = x ∈ S ;
    card(∅) = 0 ;
    x ∈ S = false  ⟹  card(ins(x, S)) = succ(card(S)) ;
    pickout(∅) = ∅ ;
    S ≠ ∅  ⟹  card(pickout(S)) = pred(card(S)) ;
    x ∈ pickout(S) = true ⟹ x ∈ S = true ;
    where :   S : Set ;   x, y : Elem ;
end SET-WITH-PICKOUT .
```

Figure 3: Yet another specification of sets

- *ins* is a constant function on $\{true{+}false\}$ which always returns $\{true{+}false\}$ itself and it works as usual on the other sets.
- $\_ \in \_$ is the constant function *true* on $\{true{+}false\}$ and it works as usual on the other sets.
- $card(\{true{+}false\}) = 1$ and the cardinal of the other sets is the usual one.
- $pickout(\{true, false\}) = \{true{+}false\}$ and *pickout* on all other sets returns the empty set.

This model clearly satisfies the specification. However $\{true{+}false\}$ is not observationally equal to a standard set because there are two distinct values (*true* and *false*) which are members of it, but its cardinal is 1.

This example, together with the *remove* and *div* examples given in Section 3, show that semantic constraints are fundamental in order to reach a specification style which is really **abstract**.

Obviously, the definition of correct proof principles for modular observational specifications requires further investigation. Some combination of "observational induction w.r.t. the generators" and of "context induction" à la Hennicker [22] could prove adequate.

As a last remark, we would like to remind that, as for standard modular specifications, the hierarchical inconsistency of a given observational specification module can result either from "inconsistent axioms" or from the introduction of "new" observations on "old" sorts. However, in the framework of modular observational specifications, "inconsistent axioms" and "adding new observations on old sorts" should be interpreted with respect to the observational satisfaction relation and the specified observations ΔObs. It is clear that the "flexibility" induced by observability will prove useful for hierarchical consistency issues as well. Moreover, if it is obvious that the observations should be carefully designed, this task is made easier since they are explicitly specified.

7 Specifying adequate observations

In this section we would like to hark back to our claim that the observational semantics defined in Section 5 is powerful ("flexible") enough, and to the role of those operations who prevent some observations (such as *enum*).

Our *SET-WITH-ENUM* example (cf. Fig. 2) was used to justify the need for an observational semantics. However, one could argue that this example was a bit ad hoc, since the purpose of the *enum* operation was rather mysterious: what could be the use of an operation which never provides observable results?

In general, such operations correspond to "internal services" used to define some other operations. For instance, assume that the *LIST* module provides a *sum* operation, which computes the sum of all the natural numbers contained in a list. Assume moreover that this *sum* operation belongs to Σ_{Obs}. Then we can compute the sum of all the natural numbers contained in a set by the following term: $sum(enum(S))$.

Well, things are not that easy: the term $sum(enum(S))$ is not observable. Nevertheless, this apparent difficulty can be easily solved by defining a new operation $sigma : Set \rightarrow Nat$, in the *SET-WITH-ENUM* specification module, with the following axiom: $sigma(S) = sum(enum(S))$. It is then sufficient to specify that *sigma* belongs to Σ_{Obs} and we are done. Note that the resulting specification module remains hierarchically consistent, since the *sum* operation on lists is associative and commutative.

One could believe that the need for a new operation *sigma* exhibits some weakness of our approach. On the contrary, our point is that preventing from observing the results of some operations can be considered as some kind of a very flexible visibility control mechanism. More precisely, these operations are "internal services" who can be used to define more complex computations, and as such they are available through the whole specification. However, a "client" of any realization of the specification is not allowed to directly invoke these internal services (e.g. by the term $sum(enum(S))$), but should instead invoke some explicitly made available service (e.g. *sigma*).

8 Conclusion

We have investigated how far modularity and observability issues can contribute to a better understanding of software correctness. We have detailed the impact of modularity on the semantics of algebraic specifications. We have shown that, with the stratified loose semantics, software correctness can be established on a module per module basis. Then we have discussed observability issues. In particular, we have explained why a behavioural semantics of observability (based on an equivalence relation between algebras) is not fully satisfactory. Therefore, we have introduced an observational semantics (based on a redefinition of the satisfaction relation) where sort observation is refined by specifying that some operations do not allow observations. Then we have integrated the stratified loose approach and our observational semantics together. As a result, we have obtained a framework (modular observational specifications) where the definition of software correctness is adequate, i.e. fits with actual software correctness. Moreover, we have shown that, with modular observational specifications, we reach a specification style which is really abstract. Our definition of software correctness is a first step towards putting software correctness proofs in practice. A promising area for further investigations is the development of (modular) proof methods on top of our approach.

Acknowledgement: We would like to thank Teodor Knapik for numerous fruitful discussions. This work is partially supported by C.N.R.S. GRECO de Programmation and E.E.C. Working Group COMPASS.

References

[1] E. Astesiano and M. Wirsing. An introduction to ASL. In *Proc. of the IFIP WG2.1 Working Conference on Program Specifications and Transformations*, 1986.

[2] F.L. Bauer et al. *The Munich project CIP. Volume I: The wide spectrum language CIP-L.* Springer-Verlag L.N.C.S. 183, 1985.

[3] G. Bernot. Good functors...are those preserving philosophy. In *Proc. of the Summer Conference on Category Theory and Computer Science*, pages 182–195, Springer-Verlag L.N.C.S. 283, 1987.

[4] G. Bernot, M. Bidoit, and C. Choppy. Abstract implementations and correctness proofs. In *Proc. of the 3rd Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 236–251, Springer-Verlag L.N.C.S. 210, 1986.

[5] G. Bernot, M. Bidoit, and T. Knapik. *Observational approaches in algebraic specifications: A comparative study.* Technical Report 6, LIENS, 1991.

[6] G. Bernot, M.-C. Gaudel, and B. Marre. Software testing based on formal specifications: A theory and a tool. *Software Engineering Journal*, 1991.

[7] M. Bidoit. Algebraic data types: Structured specifications and fair presentations. In *Proc. of the AFCET Symposium on Mathematics for Computer Science*, 1982.

[8] M. Bidoit. Development of modular specifications by stepwise refinements using the Pluss specification language. In *Proc. of the Unified Computation Laboratory*, Oxford University Press, 1991.

[9] M. Bidoit. Pluss, un langage pour le développement de spécifications algébriques modulaires. Thèse d'Etat, Université Paris-Sud, 1989.

[10] M. Bidoit. The stratified loose approach: A generalization of initial and loose semantics. In *Recent Trends in Data Type Specification, Selected Papers of the 5th Workshop on Specifications of Abstract Data Types*, pages 1–22, Springer-Verlag L.N.C.S. 332, 1987.

[11] M. Bidoit, M.-C. Gaudel, and A. Mauboussin. How to make algebraic specifications more understandable? an experiment with the Pluss specification language. *Science of Computer Programming*, 12(1), 1989.

[12] H. Ehrig, W. Fey, and H. Hansen. *ACT ONE: an algebraic specification language with two levels of semantics.* Technical Report 83–03, TU Berlin FB 20, 1983.

[13] H. Ehrig, H.-J. Kreowski, B. Mahr, and P. Padawitz. Algebraic implementation of abstract data types. *Theoretical Computer Science*, 20:209–263, 1982.

[14] H. Ehrig and B. Mahr. *Fundamentals of algebraic specification 1. Equations and initial semantics.* Volume 6 of *EATCS Monographs on Theoretical Computer Science*, Springer-Verlag, 1985.

[15] K. Futatsugi, J.A. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *Proc. of the 12th ACM Symposium on Principles of Programming Languages (POPL)*, pages 52–66, 1985.

[16] H. Ganzinger. Parameterized specifications: Parameter passing and implementation with respect to observability. *ACM Transactions on Programming Languages and Systems*, 5(3):318–354, 1983.

[17] M.-C. Gaudel and T. Moineau. A theory of software reusability. In *Proc. of the European Symposium on Programming (ESOP)*, pages 115–130, Springer-Verlag L.N.C.S. 300, 1988.

[18] V. Girratana, F. Gimona, and U. Montanari. Observability concepts in abstract data type specification. In *Proc. of Mathematical Foundations of Computer Science (MFCS)*, pages 576–587, Springer-Verlag L.N.C.S. 45, 1976.

[19] J.A. Goguen and R.M. Burstall. Introducing institutions. In *Proc. of the Workshop on Logics of Programming*, pages 221–256, Springer-Verlag L.N.C.S. 164, 1984.

[20] J.A. Goguen, J.W. Thatcher, and E.G. Wagner. *An initial approach to the specification, correctness, and implementation of abstract data types.* Volume 4 of *Current Trends in Programming Methodology*, Prentice Hall, 1978.

[21] J.V. Guttag, J.J. Horning, and J.M. Wing. *Larch in five easy pieces.* Technical Report 5, Digital Systems Research Center, 1985.

[22] R. Hennicker. *Context induction: A proof principle for behavioural abstractions and algebraic implementations.* Technical Report MIP–9001, Fakultät für Mathematik und Informatik, Universität Passau, 1990.

[23] R. Hennicker. Implementation of parameterized observational specifications. In *Proc. of TAPSOFT*, pages 290–305, Springer-Verlag L.N.C.S. 351, 1989.

[24] S. Kamin. Final data types and their specification. *ACM Transactions on Programming Languages and Systems*, 5(1):97–123, 1983.

[25] S. Mac Lane. *Categories for the working mathematician.* Volume 5 of *Graduate Texts in Mathematics*, Springer-Verlag, 1971.

[26] J. Meseguer and J.A. Goguen. *Initiality, induction and computability*, pages 459–540. *Algebraic Methods in Semantics*, Cambridge University Press, 1985.

[27] L.S. Moss, J. Meseguer, and J.A. Goguen. Final algebras, cosemicomputable algebras and degrees of unsolvability. In *Proc. of Category Theory and Computer Science*, pages 158–181, Springer-Verlag L.N.C.S. 283, 1987.

[28] L.S. Moss and S.R. Thate. Generalization of final algebra semantics by relativization. In *Proc. of the 5th Mathematical Foundations of Programming Semantics International Conference*, pages 284–300, Springer-Verlag L.N.C.S. 442, 1989.

[29] P. Nivela and F. Orejas. Initial behaviour semantics for algebraic specification. In *Recent Trends in Data Type Specification, Selected Papers of the 5th Workshop on Specification of Abstract Data Types*, pages 184–207, Springer-Verlag L.N.C.S. 332, 1987.

[30] H. Reichel. Behavioural validity of conditional equations in abstract data types. In *Contributions to General Algebra 3, Proc. of the Vienna Conference*, 1984.

[31] D. Sannella and A. Tarlecki. On observational equivalence and algebraic specification. In *Proc. of TAPSOFT*, pages 308–322, Springer-Verlag L.N.C.S. 185, 1985.

[32] D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specification revisited. *Acta Informatica*, (25):233–281, 1988.

[33] D.T. Sannella and A. Tarlecki. Building specifications in an arbitrary institution. In *Proc. of the International Symposium on Semantics of Data Types*, Springer-Verlag L.N.C.S. 173, 1984.

[34] Oliver Schoett. *Data abstraction and the correctness of modular programming.* PhD thesis, University of Edinburg, 1987.

[35] N.W.P. van Dieppen. Implementation of modular algebraic specifications. In *Proc. of the European Symposium on Programming (ESOP)*, pages 64–78, Springer-Verlag L.N.C.S. 300, 1988.

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

[37] M. Wirsing. *Structured Algebraic specifications: A kernel language.* PhD thesis, Techn. Univ. Munchen, 1983.

[38] M. Wirsing and M. Broy. Abstract data types as lattices of finitely generated models. In *Proc. of the 9th Symposium on Mathematical Foundations of Computer Science (MFCS)*, 1980.