Horizontal composability revisited*

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 Goguen and Burstall's 1980 CAT paper and its powerful influence on theories of specification implementation that were emerging at about the same time, via the introduction of the notions of *vertical* and *horizontal composition* of implementations. We then give a different view of implementation which we believe provides a more adequate reflection of the rather subtle interplay between implementation, specification structure and program structure.

1 Introduction

Goguen and Burstall's CAT paper [GB80] is surely the most influential paper in the algebraic specification literature never to be properly published in a workshop or conference proceedings or in a journal. The topic of the paper was the notion of specification implementation—also known as refinement—as a relation on specifications, used for the step-by-step development of a program from a specification of requirements. We write $SP \rightsquigarrow SP'$ to denote that SP' is an implementation of SP, with the informal meaning that SP' captures all the requirements expressed by SP but in a way that incorporates more design decisions and is thus closer to being a program. A hot question at the time was how to properly formalise this intuition. Earlier work that was relevant to this question was Hoare's work on data refinement [Hoa72] which had been incorporated into VDM [Jon80], and Milner's work on simulations [Mil71]; first approaches in the algebraic specification literature were [GTW78] and (early versions of) [Ehr82] and [EKMP82].

The main contribution of [GB80] was to sketch a compelling two-dimensional view of implementations, with implementations composing both vertically and horizontally. Composition along the vertical dimension corresponds to composition of consecutive implementations: if $SP \rightsquigarrow SP'$ and also $SP' \rightsquigarrow SP''$, then one would expect to have $SP \rightsquigarrow SP''$. This justifies the correctness of the principle of *stepwise refinement.* (This was called *vertical* composition because Goguen and Burstall drew their implementations vertically, with SP at the top; we draw them horizontally here, except in a few diagrams, to save

* This work was funded in part by the European IST FET programme under the IST-2005-015905 MOBIUS and IST-2005-016004 SENSORIA projects, and by the British–Polish Research Partnership Programme.

space.) Horizontal composition is about composing implementations of parts of a specification to give an implementation of the whole: if $SP_1 \rightsquigarrow SP'_1$ and $SP_2 \rightsquigarrow SP'_2$, then one would expect to have $SP_1 \oplus SP_2 \rightsquigarrow SP'_1 \oplus SP'_2$ for any specification-building operation $\oplus$. In particular, this should hold for composition of parameterised specifications: if $P_1 \rightsquigarrow P'_1$ and $P_2 \rightsquigarrow P'_2$ then one would expect to have $P_1;P_2 \rightsquigarrow P'_1;P'_2$. Finally, it was suggested that vertical and horizontal composition should satisfy the *double law*, which says that given a diagram of implementations admitting both vertical and horizontal composition of implementations, the result is the same whether vertical composition is done before or after horizontal composition.

In Section 2 we recall this work. A vertical composition theorem was the main result in many accounts of implementations that were emerging at about the same time, sometimes under more or less restrictive conditions on the specifications or implementations in question. Horizontal composition proved more elusive; in most cases it remained a topic for the "Future Research" section. Recent approaches go further. For instance, [GT00] (cf. [Gog96]) provides some algebraic laws that link vertical and horizontal structure, but with what seems to be a somewhat different understanding of the vertical dimension. Another example is [LF97] where horizontal composability is achieved for colimits of specification diagrams in the context of specifications for reactive systems. Still, to our knowledge, no theory of implementations ever entirely fulfilled the dream of CAT.

In Section 4 we give a different view of implementations which we believe properly reflects the subtle interplay between implementations, specification structure and program structure, and observe that it trivially satisfies a vertical composition theorem. In Section 5 we consider horizontal composition, and conclude that it does not hold in general but neither is it desirable. The problem with horizontal composition arises from the lack of correspondence in general between the structure of a specification and the structure of a program that implements it, and the difference between operations for combining specifications on one hand and operations for combining program components on the other.

2 CAT

The CAT paper [GB80] outlines a vision for a future interactive programming system to be used for the development and maintenance of programs from specifications, in which program components were to be equipped with specifications of their properties. The processes by which implementations are carried out were to be fully modularised and parameterised, and all concepts in CAT were to have a full semantic definition in order to support formal proofs of correctness. Complete system designs were to be obtained by composing a number of implementations, each one expressing an elementary design decision. Such a degree of formalization and modularization would be useful not just for achieving correctness but also for restricting the scope of re-checking required when the system is modified subsequently. Scherlis and Scott's Inferential Programming paper [SS83], which led to the Ergo project at CMU [LPRS88], contains some more detailed ideas along similar lines.

The important part of [GB80] is only a few pages long, sandwiched between a quick review of the features of the then brand-new Clear specification language [BG80] and a long OBJ definition that is only marginally relevant (see [GT79] for a presentation of OBJ as it was at the time). The key insight is the recognition of a distinction between so-called *vertical* and *horizontal* structure:

> "One basic intuition behind Cat is that the *process* of implementing a large program from its specification has a two-dimensional structure. One dimension of structure, the *horizontal*, corresponds to the structure of the specification. The second dimension, the *vertical*, corresponds to the sequence of successive refinements of the specification into actual code; the specification is at the *top*, and the code is at the *bottom*. ... A major purpose of the Cat project is to render this intuition much more precise." J.A. Goguen and R.M. Burstall [GB80]

In elaborating this point, Goguen and Burstall make reference mainly to the structure of specifications arising from parameterised specifications, known as *theory procedures* in Clear, which provide a specification of requirements that any actual parameter needs to satisfy as well as a specification of the result. Implementation of one such procedure P by another one P' having the same "metasource" and "metatarget" specifications SP and SP' respectively (where any actual argument specification must extend SP and then the result will extend SP' in a corresponding way) would be represented by the following diagram:

$$\begin{tikzcd} SP \arrow[r, bend left=60, "P"] \arrow[r, bend right=60, "P'"'] \arrow[r, phantom, "\alpha\rightsquigarrow"] & SP' \end{tikzcd}$$

where α gives the relationship between P and P'.

Nowadays the authors would presumably agree with us (see e.g. [Gog96]) that the proper entities here are *specifications of parameterised programs*, see [SST92], that is, descriptions of functions mapping algebras to algebras, rather than Clear theory procedures which map specifications (descriptions of classes of algebras) to specifications. See Section 3.

Such implementations should compose both vertically and horizontally. Horizontal composition of implementations refers to composition of implementations of parts of a specification to give an implementation of the whole. Given the following diagram:

$$\begin{tikzcd} SP \arrow[r, bend left=60, "P"] \arrow[r, bend right=60, "P'"'] \arrow[r, phantom, "\alpha\rightsquigarrow"] & SP' \arrow[r, bend left=60, "Q"] \arrow[r, bend right=60, "Q'"'] \arrow[r, phantom, "\beta\rightsquigarrow"] & SP'' \end{tikzcd}$$

horizontal composition would give

$$\begin{tikzcd}[column sep=huge] SP \arrow[r, bend left=50, "P;Q"] \arrow[r, bend right=50, "P';Q'"'] \arrow[r, phantom, "\alpha\cdot\beta \Downarrow"] & SP'' \end{tikzcd}$$

where "·" denotes horizontal composition of implementations and ";" stands for composition of specifications of parameterised programs. The same idea applies to other specification-building operations: given $\alpha : SP_1 \rightsquigarrow SP_1'$ and $\alpha' : SP_2 \rightsquigarrow SP_2'$, one would expect to have $\alpha \odot \alpha' : SP_1 \oplus SP_2 \rightsquigarrow SP_1' \oplus SP_2'$ for any specification-building operation $\oplus$. This depends on having an operation $\odot$ for combining implementations that corresponds to each operation $\oplus$ for combining specifications. But according to [GB80]:

> "Questions remain about how the Clear operations can be extended from specifications to implementations."

Vertical composition of implementations corresponds to stepwise refinement:

$$\begin{tikzcd}[column sep=huge] SP \arrow[r, bend left=60, "P"] \arrow[r, "P'"] \arrow[r, bend right=60, "P''"'] \arrow[r, phantom, shift left=3ex, "\alpha \Downarrow"] \arrow[r, phantom, shift right=3ex, "\alpha' \Downarrow"] & SP' \end{tikzcd} \qquad \begin{tikzcd}[column sep=huge] SP \arrow[r, bend left=60, "P"] \arrow[r, bend right=60, "P''"'] \arrow[r, phantom, "\alpha;\alpha' \Downarrow"] & SP' \end{tikzcd}$$

The composed implementation $\alpha;\alpha'$ combines the design decisions in α with those in α': for instance, if α shows how to implement graphs using sets, and α' shows how to implement sets using lists, then $\alpha;\alpha'$ shows how to implement graphs using lists.

Now, suppose we have a structured specification with consecutive implementations of its components, like so:

$$\begin{tikzcd}[column sep=huge] SP \arrow[r, bend left=60, "P"] \arrow[r, "P'"] \arrow[r, bend right=60, "P''"'] \arrow[r, phantom, shift left=3ex, "\alpha \Downarrow"] \arrow[r, phantom, shift right=3ex, "\alpha' \Downarrow"] & SP' \arrow[r, bend left=60, "Q"] \arrow[r, "Q'"] \arrow[r, bend right=60, "Q''"'] \arrow[r, phantom, shift left=3ex, "\beta \Downarrow"] \arrow[r, phantom, shift right=3ex, "\beta' \Downarrow"] & SP'' \end{tikzcd}$$

In this situation we may apply vertical composition to give implementations $\alpha;\alpha'$ and $\beta;\beta'$, and then apply horizontal composition to give an implementation $(\alpha;\alpha')\cdot(\beta;\beta') : P;Q \rightsquigarrow P'';Q''$. Alternatively, we may first apply horizontal composition to give implementations $\alpha\cdot\beta$ and $\alpha'\cdot\beta'$, and then apply vertical composition to give an implementation $(\alpha\cdot\beta);(\alpha'\cdot\beta') : P;Q \rightsquigarrow P'';Q''$. Goguen and Burstall conjecture that these two implementations should be *the same*: the order of composition should not matter. If this "double law" holds then implementations form a *two-dimensional category*, see [Mac71] (where the double law is called the "interchange law"). They speculate that the double law may not hold for some specification-building operations, and then extra care must be taken at such points during the implementation process.

All of this discussion is set in the context of an arbitrary institution [GB92]—a concept which first appeared in the semantics of CLEAR [BG80]—abstracting away from the particular logical system used to write specifications. There is no formal definition of what implementation of specifications means. Goguen and Burstall also suggest that the CAT framework would be appropriate for use with various different programming languages and programming paradigms. Although functional languages are the most obvious fit, they speculate that the use of imperative languages and assembly languages should not pose any insurmountable obstacles.

3 Specifications and programs

The precise syntax of specifications is not very important in this paper. More significant is the way that the semantics of specifications is defined: for each specification SP, we define its signature $Sig(SP)$ and its class of models, $Mod(SP)$, where each SP-model is a $Sig(SP)$-algebra: $Mod(SP) \subseteq Alg(Sig(SP))$. The signature of a specification defines an interface giving names to the required program components, while its models represent programs that are considered to be its correct realizations. If $Sig(SP) = \Sigma$ we will say that SP is a *Σ-specification.*

The framework we are describing is independent of any particular institution [GB92]. It can therefore be used with different programming paradigms by selecting a notion of model that reflects the features of the paradigm in question. However, for the sake of concreteness and simplicity let us concentrate on standard many-sorted algebras over standard algebraic signatures, specified using axioms in first-order logic with equality. These capture a subset of Standard ML programs (so-called *structures*) over Standard ML signatures [MTHM97], comprising first-order non-polymorphic datatypes and first-order non-polymorphic properly-terminating functions.

Example 3.1. The following signature defines an interface for a program to sort lists of elements with respect to an order relation on the type of elements:

```
signature SORTELEM =
  sig
    type elem
    val ord : elem * elem -> bool
    type listelem
    val nil : listelem
    val cons : elem * listelem -> listelem
    val sort : listelem -> listelem
  end
```

A structure over this signature provides code for the required components, including such a sorting program:

```
structure SortElem : SORTELEM =
  struct
    type elem = int
    fun ord(x,y) = x >= y
    datatype listelem = nil | cons of elem * listelem
    fun sort l = ... (* code for sort *) ...
  end
```

The semantics of Standard ML [MTHM97] can be used to interpret the above code as a definition of an algebraic signature, call it $[\![\texttt{SORTELEM}]\!]$, and a particular algebra over this signature $[\![\texttt{SortElem}]\!] \in \mathit{Alg}([\![\texttt{SORTELEM}]\!])$.

Example 3.2. The following specification has the above program as a correct realization:

```
specification SORTELEMSPEC =
  spec
    type elem
    val ord : elem * elem -> bool
    axiom ... (* ord is transitive, reflexive and antisymmetric *) ...

    datatype listelem = nil | cons of elem * listelem
    val sort : listelem -> listelem
    axiom ... (* sort produces a permutation of its input *) ...
    axiom ... (* the output of sort is ordered according to ord *) ...
  end
```

Then $\mathit{Sig}(\texttt{SORTELEMSPEC}) = [\![\texttt{SORTELEM}]\!]$ and $[\![\texttt{SortElem}]\!] \in \mathit{Mod}(\texttt{SORTELEMSPEC}) \subseteq \mathit{Alg}([\![\texttt{SORTELEM}]\!])$.

For the sake of example, one often considers the following rudimentary ways of building specifications:

***basic specifications*:** For any signature Σ and set Φ of Σ-sentences, the *basic specification* $\langle \Sigma, \Phi \rangle$ is a Σ-specification with $\mathit{Mod}(\langle \Sigma, \Phi \rangle) = \{M \in \mathit{Alg}(\Sigma) \mid M \models \Phi\}$. (SORTELEMSPEC above is a basic specification.)

***union*:** For any Σ, given Σ-specifications SP_1 and SP_2, their *union* $SP_1 \cup SP_2$ is a Σ-specification with $\mathit{Mod}(SP_1 \cup SP_2) = \mathit{Mod}(SP_1) \cap \mathit{Mod}(SP_2)$.

***translation*:** For any signature morphism $\sigma \colon \Sigma \to \Sigma'$ and Σ-specification SP, **translate** SP **by** σ is a Σ'-specification with $\mathit{Mod}(\textbf{translate } SP \textbf{ by } \sigma) = \{M' \in \mathit{Alg}(\Sigma') \mid M'|_\sigma \in \mathit{Mod}(SP)\}$.[1]

***hiding*:** For any $\sigma \colon \Sigma \to \Sigma'$ and Σ'-specification SP', **derive from** SP' **by** σ is a Σ-specification with $\mathit{Mod}(\textbf{derive from } SP' \textbf{ by } \sigma) = \{M'|_\sigma \mid M' \in \mathit{Mod}(SP')\}$.[1]

[1] For any signature morphism $\sigma \colon \Sigma \to \Sigma'$ and algebra $M' \in \mathit{Alg}(\Sigma')$, $M'|_\sigma \in \mathit{Alg}(\Sigma)$ is the *reduct* of M' with respect to σ, see e.g. [ST99]. When σ is a signature inclusion, $M'|_\sigma$ may be written as $M'|_\Sigma$.

This follows ASL [SW83,ST88a] and is different from CLEAR, where specification expressions denoted *theories* which in turn have model classes, see [ST97] for a discussion of the difference. The operations are more primitive but are similarly expressive: for instance "+" in CLEAR corresponds to union of suitably translated specifications over different signatures, where the translations respect shared subspecifications.

This defines a number of so-called *specification-building operations* which map specifications to more complex specifications: we have constant specification-building operations (basic specifications), one binary specification-building operation (union) and two unary ones (translation and hiding). In fact, each of these may be viewed as a family of operations, indexed by signatures (union) and specification morphisms (translation and hiding). Once this "static" indexing is fixed, each specification-building operation semantically amounts to a function on appropriate classes of models.

One property of the above specification-building operations will prove crucial for further considerations: an n-ary specification-building operation op is *monotone* if it is monotone as a function on model classes. That is: for any specifications SP_1, SP'_1, $\ldots$, SP_n, SP'_n, such that $Sig(SP_i) = Sig(SP'_i)$ and $Mod(SP_i) \subseteq Mod(SP'_i)$ for $i = 1, \ldots, n$, we also have $Mod(op(SP_1, \ldots, SP_n)) \subseteq Mod(op(SP'_1, \ldots, SP'_n))$.

All the above specification-building operations, and therefore any operation that may be defined using them, are monotone. In fact, nearly all specification-building operations one may find in the literature are monotone. The only exception we are aware of are operations that select initial or free models of specifications—one may argue though that such an operation should be viewed as simply imposing an additional *constraint* on the class of models of a specification, like an axiom, rather than as specification-building operations in their own right (see for instance *data constraints* in [GB92]).

Structured specifications in CASL [BM04,CoF04] are based on the operations above as well; somewhat more convenient notation is introduced there, which we will use in examples too. For instance, union (not limited to specifications with identical signatures) is written with `and`, translation along surjective signature morphisms is written with `with` (followed by the mapping of symbols), hiding is written with `reveal` or `hide` (followed by a list of symbols). Perhaps most useful is `then`, which is an obvious combination of a translation along a signature inclusion with union to build an *extension* of a specification by new sorts, operations and/or axioms.

Example 3.3. Here are some examples of structured specifications:

```
specification ELEMSPEC =
  spec
    type elem
    val ord : elem * elem -> bool
    axiom ... (* ord is transitive, reflexive and antisymmetric *) ...
  end
specification ELEMLISTSPEC =
```

```
  ELEMSPEC then
    datatype listelem = nil | cons of elem * listelem
  end
specification PERMELEMSPEC =
  ELEMLISTSPEC then
    val perm : listelem -> listelem
    axiom ... (* perm produces a permutation of its input *) ...
  end
specification ORDERELEMSPEC =
  ELEMLISTSPEC then
    val order : listelem -> listelem
    axiom ... (* the output of order is ordered w.r.t. ord *) ...
  end
specification STRUCTSORTELEMSPEC =
    {PERMELEMSPEC with perm |-> sort}
    and
    {ORDERELEMSPEC with order |-> sort}
```

Specifications `SORTELEMSPEC` of Example 3.2 and `STRUCTSORTELEMSPEC` above are equivalent: they have the same signature ($[\![$`SORTELEM`$]\!]$ in both cases, see Example 3.1) and the same class of models.

In common with all work on algebraic specification we have taken the view that algebras model programs. But in general we are interested in program *components* which define new sorts and operations in terms of some existing ones. These may be *generic* components, where the parameters are supplied explicitly, or components that explicitly import or implicitly build on other components. In each case, we need to model components as functions mapping algebras to algebras; in the case of explicit or implicit imports this reflects the way that the newly-defined sorts and operations depend on the imports.

Definition 3.4. *Let Σ and Σ' be signatures. A* $(\Sigma \to \Sigma')$-constructor *is a function*[2] *mapping Σ-algebras to Σ'-algebras.*

In the standard algebraic institution, constructors correspond most directly to Standard ML *functors* defining first-order non-polymorphic datatypes and first-order non-polymorphic properly-terminating functions, where the input and output signatures are explicit.

Example 3.5. Here is an example of a constructor in Standard ML:

```
signature ELEM =
sig
  type elem
  val ord : elem * elem -> bool
```

[2] In general, we need to consider *partial* constructors, where the result may not be defined for every algebra over the parameter signature but only for those that satisfy additional constraints. See [ST89]. For simplicity, we restrict attention to total constructors here, with a few comments in footnotes concerning partial constructors.

```
end

functor Sort(X: ELEM) : SORTELEM =
  struct
    open X
    datatype listelem = nil | cons of elem * listelem
    fun sort l = ... (* code for sort *) ...
  end
```

The semantics of Standard ML can be used to interpret the above code as defining a function mapping $[\![\mathtt{ELEM}]\!]$-algebras to $[\![\mathtt{SORTELEM}]\!]$-algebras, i.e. an $([\![\mathtt{ELEM}]\!] \to [\![\mathtt{SORTELEM}]\!])$-constructor. One important property of this function is that it is *persistent*: the argument structure is extended to the result structure, preserving the interpretation of parameter types and values.

Any $(\Sigma \to \Sigma')$-constructor κ determines a specification-building operation, written κ as well, that takes any Σ-specification SP to a Σ'-specification having the image of $Mod(SP)$ under κ as its models: $Mod(\kappa(SP)) = \{\kappa(M) \mid M \in Mod(SP)\}$. Hiding is one such specification-building operation, determined by reduct. The other specification-building operations discussed above do not arise in such a way, in general. Translation is determined by a total constructor only when it is with respect to a bijective renaming[3], and then it coincides with hiding with respect to the inverse of that renaming. CASL union is not determined by a total constructor unless there is no overlap ("sharing") between the signatures of the arguments.[4]

Constructors may themselves be specified. For the same reason as ordinary specifications describe classes of algebras, constructor specifications describe classes of constructors, that is, classes of functions mapping algebras to algebras [SST92].

Definition 3.6. *Given specifications SP and SP', the* constructor specification *$SP \to SP'$ specifies the class of $(Sig(SP) \to Sig(SP'))$-constructors that map models of SP to models of SP': $Mod(SP \to SP') = \{F\colon Alg(Sig(SP)) \to Alg(Sig(SP')) \mid$ for each $A \in Mod(SP), F(A) \in Mod(SP')\}$.*[5]

Moreover, when $Sig(SP)$ overlaps with $Sig(SP')$ then the specified constructors should preserve the interpretation of the overlapping sorts and operations. In particular, when $Sig(SP)$ is a subsignature of $Sig(SP')$, then as in CASL *we require the functions in $Mod(SP \to SP')$ to be* persistent*: when $F\colon Alg(Sig(SP)) \to Alg(Sig(SP')) \in Mod(SP \to SP')$ then for every model $A \in Mod(SP)$, $F(A) \in Mod(SP')$ is such that $F(A)|_{Sig(SP)} = A$.*

[3] Translations along surjective signature morphisms are determined by partial constructors, in general.

[4] When there is overlap, CASL union is determined by a partial constructor which amalgamates models that coincide on the shared subsignature.

[5] If partial constructors are considered, an additional requirement here would be that their domain contains $Mod(SP)$.

Example 3.7. Recall Examples 3.1–3.3. Then $\mathtt{ELEMSPEC} \to \mathtt{SORTELEMSPEC}$ is a specification of (persistent) constructors $F\colon Alg([\![\mathtt{ELEM}]\!]) \to Alg([\![\mathtt{SORTELEM}]\!])$ that when given a model $E \in Mod(\mathtt{ELEMSPEC})$ extends it to a model $F(E) \in Mod(\mathtt{SORTELEMSPEC})$. One example of such a constructor is the functor $\mathtt{Sort} \in Mod(\mathtt{ELEMSPEC} \to \mathtt{SORTELEMSPEC})$, presented in Example 3.5. Constructor specifications correspond to functor specifications in Extended ML, see [KST97].

The generalisation to n-ary constructors and constructor specifications is straightforward.

4 Implementations and vertical composition

A very simple notion of specification implementation is the following:

Definition 4.1. *Let SP and SP' be specifications such that $Sig(SP) = Sig(SP')$. Then SP' is a* simple implementation *of SP, written $SP \rightsquigarrow SP'$, if $Mod(SP) \supseteq Mod(SP')$.*

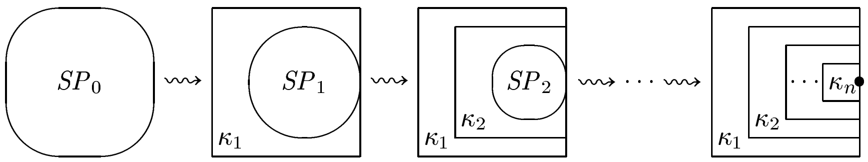
This simply requires that all of the correct realizations of SP' are correct realizations of SP. That is, SP' incorporates all the requirements that are in SP, and perhaps other constraints that result from additional design decisions.

For simplicity, the definition of simple implementation requires the signatures of both specifications to be the same. The hiding operation may be used to adjust the signatures (for example, by removing auxiliary functions from the signature of the implementing specification) if this is not the case.

The fact that simple implementations vertically compose is an immediate consequence of the transitivity of the subset relation:

Proposition 4.2. *If $SP \rightsquigarrow SP'$ and $SP' \rightsquigarrow SP''$ then $SP \rightsquigarrow SP''$.*

The notion of simple implementation is powerful enough (in the context of a sufficiently rich specification language) to handle all concrete examples of interest. However, it is not very convenient. During the process of developing a program, the successive specifications incorporate more and more details arising from successive design decisions. Thereby, some parts become fully determined, and remain unchanged as a part of the specification until the final program is obtained. The following diagram is a visual representation of this situation, where $\kappa_1, \ldots, \kappa_n$ label the parts that become determined at consecutive steps.



It is more convenient to avoid such clutter by separating the finished parts from the specification, putting them aside, and proceeding with the development of the unresolved parts only:

$$SP_0 \underset{\kappa_1}{\rightsquigarrow} SP_1 \underset{\kappa_2}{\rightsquigarrow} SP_2 \underset{\kappa_3}{\rightsquigarrow} \cdots \underset{\kappa_n}{\rightsquigarrow} \bullet\ SP_n = \textsc{Empty}$$

where EMPTY is a specification for which a standard implementation *empty* is available.

It is important for the finished parts $\kappa_1, \ldots, \kappa_n$ to be independent of the particular choice of realization for what is left: they should extend *any* realization of the unresolved part to a realization of what is being implemented. This is exactly what is required by the notion of a *constructor* defined in Sect. 3: κ_i is a *function* taking models of SP_i to models of SP_{i-1}. These considerations motivate a more elaborate version of the notion of implementation:

Definition 4.3 ([ST88b]). *Given specifications SP and SP' and constructor $\kappa : Alg(Sig(SP')) \to Alg(Sig(SP))$, we say that SP' is a* constructor implementation of SP via κ, *written $SP \underset{\kappa}{\rightsquigarrow} SP'$, if $\kappa \in Mod(SP' \to SP)$.*

Thus, in the development diagram above, $\kappa_i\colon Alg(Sig(SP_i)) \to Alg(Sig(SP_{i-1}))$ with $\kappa_i \in Mod(SP_i \to SP_{i-1})$ for $1 \le i \le n$; that is, each κ_i corresponds to a parameterised program with input interface SP_i and output interface SP_{i-1}. Given a model M of SP_i, κ_i may be applied to yield a model $\kappa_i(M)$ of SP_{i-1}.

Example 4.4. From Example 3.7, we have `SORTELEMSPEC` $\underset{\texttt{Sort}}{\rightsquigarrow}$ `ELEMSPEC`. That is, the task of implementing sorting of lists of elements with respect to a function `ord` is reduced by means of the constructor `Sort` to the task of implementing `elem` and `ord`.

The definition of constructor implementation generalises smoothly to implementations of constructor specifications. This requires *higher-order* constructors; for details see [ST97].

It is easy to see that constructor implementations compose vertically:

Proposition 4.5. *If $SP \underset{\kappa}{\rightsquigarrow} SP'$ and $SP' \underset{\kappa'}{\rightsquigarrow} SP''$ then $SP \underset{\kappa';\kappa}{\rightsquigarrow} SP''$.*

So, a constructor implementation via $\kappa\colon Alg(Sig(SP')) \to Alg(Sig(SP))$ composed with a constructor implementation via $\kappa'\colon Alg(Sig(SP'')) \to Alg(Sig(SP'))$ yields a constructor implementation via $\kappa';\kappa\colon Alg(Sig(SP'')) \to Alg(Sig(SP))$, which is just the composition of the functions κ' and κ written in diagrammatical order.

Once the development process is finally complete (that is, when nothing is left unresolved, as in the diagram above) we can successively apply the constructors to obtain a correct realization of the original specification. The correctness of the final outcome follows from the correctness of the individual constructor implementation steps via vertical composition.

Proposition 4.6. *Given a chain of constructor implementation steps*

$$SP_0 \underset{\kappa_1}{\rightsquigarrow} SP_1 \underset{\kappa_2}{\rightsquigarrow} \cdots \underset{\kappa_n}{\rightsquigarrow} SP_n = \textsc{Empty}$$

we have $(\kappa_n;\cdots;\kappa_2;\kappa_1)(empty) \in Mod(SP_0)$.

Many approaches to implementation in the literature make use of a restrictive kind of constructor defined by a parameterised program having a particular rigid form: for example, the notion of implementation in [EKMP82] corresponds to the use of a constructor obtained by composing a free construction with a reduct, then a restriction to a subalgebra, and finally a quotient, in that order. Then the vertical composition of two implementations is required to yield an implementation of the same form, which is only possible under certain additional conditions on the specifications involved. This amounts to a requirement that the composition of parameterised programs be forced into some given normal form, which corresponds to requiring programs to be written in a rather restricted programming language.

5 Horizontal composition

In Sect. 3 we have recalled a few basic specification-building operations, which form the backbone of many *specification languages*. Since the pioneering work on Clear [BG80], a number of such languages have been designed and used, with Casl [BM04,CoF04] as a prime recent example. They all aim at providing a convenient way to build specifications in a structured manner, where specification-building operations are used to gradually construct more and more complex specifications out of simpler component specifications. This *horizontal structure* of specifications (in the terminology of [GB80]) is indispensable for facilitating the understanding and use of any practical (hence: large and complex) specification. Typical ways in which the horizontal structure of specifications has been successfully exploited include the compositional semantics of complex specifications languages like Casl [BCH+04] and compositional proof systems for consequences of specifications, as introduced in [ST88a] and analyzed in [Bor02], even if for practical specification languages compositionality may sometimes be sacrified [MHAH04].

Under a mild assumption of monotonicity of the specification-building operations involved, the horizontal structure of specifications may also be exploited in the development process:

Proposition 5.1. *Suppose that op is a monotone n-ary specification-building operation. If $SP_1 \rightsquigarrow SP'_1$, $\ldots$, $SP_n \rightsquigarrow SP'_n$ then $op(SP_1, \ldots, SP_n) \rightsquigarrow op(SP'_1, \ldots, SP'_n)$.*

For simple implementations, Prop. 5.1 captures the essence of horizontal composition, as introduced in [GB80]. For constructor implementations this takes the following form:

Proposition 5.2. *Suppose that op is a monotone n-ary specification-building operation. If $SP_1 \underset{\kappa_1}{\rightsquigarrow} SP'_1$, $\ldots$, $SP_n \underset{\kappa_n}{\rightsquigarrow} SP'_n$ then $op(SP_1, \ldots, SP_n) \rightsquigarrow op(\kappa_1(SP'_1), \ldots, \kappa_n(SP'_n))$.*

Note that κ_1 in $\kappa_1(SP'_1)$ refers to the specification-building operation determined by the constructor κ_1—see Sect. 3—and similarly for the other constructors.

The strength and usefulness of Props. 5.1 and 5.2 are severely limited by two fundamental problems.

First, the consistency of specifications is not preserved under such refinement in general. In Prop 5.1, $op(SP_1, \dots, SP_n)$ may be a perfectly implementable (consistent) specification, while $op(SP'_1, \dots, SP'_n)$ is inconsistent, and hence cannot be implemented, even if implementation of each of the refined individual component specifications SP'_1, ..., SP'_n is unproblematic.

Example 5.3. Consider the following trivial example:

```
specification EVEN =
    spec val a : int
         axiom exists k : int . a = 2 * k
    end
specification SMALL =
    spec val a : int
         axiom a > 0 andalso a < 10
    end
specification SMALL_EVEN = SMALL and EVEN
```

The last specification is formed as a union of two simpler specifications, and thus combines the requirements they impose. (Obviously, algebras in $[\![\texttt{SMALL\_EVEN}]\!]$ have $\texttt{a} \in \{2, 4, 6, 8\}$.)

Since and is monotone, Prop. 5.1 allows one to refine SMALL_EVEN by refining its component specifications independently. Consider for instance:

```
specification VERY_EVEN =
    spec val a : int
         axiom exists k : int . a = 8 * k
    end
specification VERY_SMALL =
    spec val a : int
         axiom a > 0 andalso a < 5
    end
specification VERY_SMALL_VERY_EVEN = VERY_SMALL and VERY_EVEN
```

Clearly, we have then EVEN $\rightsquigarrow$ VERY_EVEN and SMALL $\rightsquigarrow$ VERY_SMALL, and so by Prop. 5.1,

$$\texttt{SMALL\_EVEN} \rightsquigarrow \texttt{VERY\_SMALL\_VERY\_EVEN}.$$

However, even though both VERY_SMALL and VERY_EVEN are consistent and separately can be easily implemented, the specification VERY_SMALL_VERY_EVEN is inconsistent, and so taking this implementation step cannot lead to a final realization of SMALL_EVEN.

The above problem with consistency of the refined specification may arise even with a unary specification-building operation *op* (for instance, consider translation along a non-injective signature morphism). However, it does not arise if the operation *op* is determined by a constructor.

The other problem with refinement based on horizontal composability is perhaps even more fundamental. Although the horizontal structure of a specification is crucial for its understanding and use, in general this structure may well be quite different from the modular structure of the final program that implements it. The aims of horizontal structure at the level of the original, high-level, abstract requirements specification are quite separate from the aims of modular structure in the final program. An interesting and convincing example is presented in [FJ90] in a somewhat different framework, but the case study and the general line of reasoning carry over here as well. The conclusion from this is that while horizontal composability (with respect to monotone specification-building operations) yields sound refinements and so may be used when appropriate, it cannot be the only way to implement structured specifications. We need separate means to explicitly mark design decisions that fix the final modular structure of the program under development, which requires the top-level specification-building operations to be determined by constructors. Once such a *design specification* [AG97] has been fixed, this top-level horizontal structure is to be preserved in programs resulting from the development process, and further development proceeds for each component specification separately. The final result is then obtained by applying the top-level constructors to the outcomes of these separate developments.

Consider for instance an n-ary constructor *op*. Abusing slightly the notation of *architectural specifications* [BST02] as provided by CASL [BM04,CoF04], a design specification that designates the top-level constructor *op* to be preserved and used at the top level of the modular structure of the final program may take the following form:

```
arch spec OP_DESIGN =
    units U_1 : SP_1
          ...
          U_n : SP_n
    result op(U_1,...,U_n)
```

This introduces names (`U_1`, ..., `U_n`) of *units* (or *modules*) to be further developed as realizations of their specifications (`SP_1`, ..., `SP_n`, respectively) and then put together using the constructor `op` to yield the overall realization of the system.[6] An architectural specification can be compared with ordinary specifications by defining its models to be all the possible result units that may be built in this way. Then one may consider refinements involving architectural specifications, like $SP \rightsquigarrow$ `OP_DESIGN`. This captures a design decision to implement the specification SP by a modular system, where the top-level modules `U_1`, ..., `U_n`, fulfilling specifications `SP_1`, ..., `SP_n`, respectively, are put together using the constructor `op`.

In particular, we always have: $\mathtt{op}(\mathtt{SP\_1}, \ldots, \mathtt{SP\_n}) \rightsquigarrow \mathtt{OP\_DESIGN}$. Note that `op` refers here to the specification-building operation determined by the constructor `op`, see Sect. 3.

[6] If `op` is partial, it is necessary to ensure that no tuple of models which may potentially be given as an argument to `op` is outside its domain. See [BST02].

For unary constructors K, the constructor implementation $SP \underset{\mathtt{K}}{\rightsquigarrow} SP'$ corresponds exactly to the refinement $SP \rightsquigarrow$ K_DESIGN, where

```
arch spec K_DESIGN = unit U : SP' result K(U)
```

An important twist in CASL architectural specifications is that the units used here may in fact be *generic* modules, that is, constructors with specifications taking the form discussed in Sect. 3. This allows one to delegate "coding" of constructors (as, say, Standard ML functors) to further development of the corresponding units, and to limit the vocabulary of the constructors in use in the result unit expression to a few basic constructs including the application of a generic unit to an argument.

Example 5.4. Recall the specifications in Examples 3.1–3.7. Note that the specification SORTELEMSPEC requires a sorting program sort for *some* realization for the type elem and ordering predicate ord chosen by the implementor. The following architectural specification decomposes this task by separating out on one hand the task to build such a realization for elem and ord, and on the other hand, the task of providing a sorting program sort that will work for *any* such realization. The overall result is then given by instantiating the outcome of the latter task to the outcome of the former one.

```
arch spec SORT_SPEC =
  units E : ELEMSPEC
        S : ELEMSPEC -> SORTELEMSPEC
  result S(E)
```

Then SORTELEMSPEC $\rightsquigarrow$ SORT_SPEC. We also have STRUCTSORTELEMSPEC $\rightsquigarrow$ SORT_SPEC even though the structure of SORT_SPEC does not match the structure of STRUCTSORTELEMSPEC.

The main point of architectural specifications as sketched above is that further developments of the specified units may proceed independently from each other, and the final results of these developments, which fulfill the unit specifications, may then be put together as prescribed by the result unit expression. Soundness of this procedure is guaranteed by the horizontal composability of implementations, Props. 5.1 and 5.2—however, with the additional effect that consistency of the result is ensured provided that each refined component specification remains consistent.

Note that horizontal composability follows from the following properties of implementation steps involving individual component specifications. Let op be a monotone n-ary specification-building operation.

- If $SP_1 \rightsquigarrow SP'_1$ then $op(SP_1, \ldots, SP_n) \rightsquigarrow op(SP'_1, \ldots, SP_n)$.

  ...
- If $SP_n \rightsquigarrow SP'_n$ then $op(SP_1, \ldots, SP_n) \rightsquigarrow op(SP_1, \ldots, SP'_n)$.

Prop. 5.1 then follows by a simple application of vertical composability (Prop. 4.2).

Similarly, for constructor implementations we have:

– If $SP_1 \underset{\kappa_1}{\rightsquigarrow} SP'_1$ then $op(SP_1, \ldots, SP_n) \rightsquigarrow op(\kappa_1(SP'_1), \ldots, SP_n)$.
...
– If $SP_n \underset{\kappa_n}{\rightsquigarrow} SP'_n$ then $op(SP_1, \ldots, SP_n) \rightsquigarrow op(SP_1, \ldots, \kappa_n(SP'_n))$.

Prop. 5.2 now follows easily by Prop. 4.2.

The refinements of component specifications here are entirely independent from each other, and so may be taken in an arbitrary order. "Composition" of such independent refinements in any chosen order always yields the same result.

The key case here is when *op* is a constructor, and the specification considered is the architectural specification `OP_DESIGN` as above. In the notation of [MST04], refinements of individual unit specifications can be defined as follows:

```
refinement R_1 = U_1: SP_1 refined to arch spec
                                        unit X_1 : SP'_1
                                        result K_1(X_1)
...
refinement R_n = U_n: SP_n refined to arch spec
                                        unit X_n : SP'_n
                                        result K_n(X_n)
```

In [MST04], we have introduced the possibility of composing refinements, and indeed, according to the formal semantics given there, the above refinements can be composed in an arbitrary order, and each such composition yields the same result. For instance:

```
refinement R_1_to_n = R_1 then ... then R_n
refinement R_n_to_1 = R_n then ... then R_1
```

yields `R_1_to_n` $=$ `R_n_to_1`. The fact that these refinements coincide in the case `n` $= 2$ captures the "double law" of [GB80], see Sect. 2.

In fact, [MST04] provides for the possibility of writing down the corresponding fragment of a development tree as follows:

```
arch spec DEVELOP =
    units U_1 : SP_1 refined to arch spec
                                unit X_1 : SP'_1
                                result K_1(X_1)
          ...
          U_n : SP_n refined to arch spec
                                unit X_n : SP'_n
                                result K_n(X_n)
    result op(U_1,...,U_n)
```

It should be clear (and this can be formally proved within the framework of [MST04]) that this is equivalent to the following architectural specification:

```
arch spec OP_DESIGN' =
    units X_1 : SP'_1
          ...
          X_n : SP'_n
    result op(K_1(X_1),...,K_n(X_n))
```

This explicitly captures the composition of the design decision to use `op` as the top-level constructor (captured by `OP_DESIGN`) with the constructor implementations for components in an arbitrary order. Note that this easily generalises to implementations of individual components that lead to further decomposition, again given by architectural specifications.

Example 5.5. Continuing Examples 5.4 and 3.1–3.7, consider the following additional specification:

```
specification INSERTELEMLISTSPEC =
  ELEMLISTSPEC then
    val insert : elem * listelem -> listelem
    axiom ... (* if l is ordered then insert(e,l) puts e into l
                             so that the result is ordered *) ...
```

Then the architectural specification `SORT_SPEC` may be refined as follows:

```
arch spec SORT_SPEC' =
  units E: ELEMSPEC
        S: ELEMSPEC -> SORTELEMSPEC
           refined to
             arch spec
               units L: ELEMSPEC -> ELEMLISTSPEC
                     I: ELEMLISTSPEC -> INSERTELEMLISTSPEC
                     IS: INSERTELEMLISTSPEC -> SORTELEMSPEC
               result lambda X: ELEMSPEC . IS(I(L(X)))
  result S(E)
```

We can also make the resulting overall design explicit as follows:

```
arch spec SORT_SPEC'' =
  units E: ELEMSPEC
        L: ELEMSPEC -> ELEMLISTSPEC
        I: ELEMLISTSPEC -> INSERTELEMLISTSPEC
        IS: INSERTELEMLISTSPEC -> SORTELEMSPEC
  result IS(I(L(E)))
```

Of course, we then have `SORTELEMSPEC` $\rightsquigarrow$ `SORT_SPEC''`. Further development may involve for instance direct implementations of the generic units `L`, `I` and `IS` as Standard ML functors, entirely independent from each other.

The above example is misleadingly simple since there is no requirement for sharing between the units involved in the design. In general this need not be the case. Suppose that the task of implementing a specification SP_{big} is decomposed into the tasks of implementing specifications SP_1 and SP_2 where $[\![SP_1 \texttt{ and } SP_2]\!] \subseteq [\![SP_{big}]\!]$ but the signatures of SP_1 and SP_2 overlap. If a realization of SP_{big} is to be obtained by combining realizations of SP_1 and SP_2, these two realizations need to share the realization of their common part. This is handled as in [Bur84]: we provide a specification SP of the common part and add its realization as a new task, and then use (persistent) generic units to separately extend the resulting unit to realizations of SP_1 and SP_2, thus ensuring that they share this common part and so can be put together.

Formalizing this: if $Sig(SP) \supseteq Sig(SP_1) \cap Sig(SP_2)$ and $[\![SP \texttt{ and } SP_1 \texttt{ and } SP_2]\!] \subseteq [\![SP_{big}]\!]$, then $SP_{big} \rightsquigarrow$ SHARING_SPEC where

```
arch spec SHARING_SPEC =
  units U: SP
        F1: SP->SP1
        F2: SP->SP2
  result F1(U) and F2(U)
```

Here, "and" is a partial binary constructor which amalgamates two models provided that they coincide on their common subsignature—see footnote 4 and note that the requirement mentioned in footnote 6 is satisfied. Note again that further refinements of the components may proceed independently from each other.

6 Conclusions

What emerged from [GB80] was a powerful and stimulating view of the process of systematic development of software from high-level formal specifications. What was insightful, new and perhaps ahead of its time then was the stress on *structure* as the only realistic means to master the size and complexity of practical software development projects.

The CAT paper identified formally two orthogonal aspects of structure in the process of software development: the vertical dimension, the structure of the development process as such; and the horizontal dimension, the structure of the specifications involved in development. Making this distinction was crucial to separating the two dimensions, for separate study, with vertical and horizontal composability as the key result to aim for. These separate lines of research resulted in a lot of interesting work, crucial for an adequate formalisation of the development process.

The vertical dimension proved easier for the theory: in spite of technical difficulties, in many frameworks the key vertical compositionality result has been established, with our composition of constructor implementations (further generalised to composition of *abstractor implementations*, not discussed here, see [ST88b,ST97]) covering the previous work as special cases—with the results recalled in Sect. 4.

The horizontal dimension attracted much work and research as well (including the pioneering work by Goguen and Burstall themselves on CLEAR [BG80]) with many specification languages designed that included various forms of horizontal structuring of specifications, and many key results on the use of this horizontal structure for proper understanding and use of large specifications. However, the interaction of the horizontal structure with development, formulated in [GB80] as horizontal composability, and the double law used to capture the interplay between the two dimensions, proved much tougher. In fact, there are hints in [GB80] which indicate that the authors viewed this idea as somewhat speculative, and foresaw potential obstacles in making it effective. We have already quoted their thought that the task to design implementation composition operations corresponding to all specification-building operations in CLEAR might be difficult. They also mention that the structure of a specification, with horizontal composition as the way to build its implementations, may constitute an "implementation bias", thus (perhaps unnecessarily) preventing implementations having a different structure. From our current perspective, it seems a bit unrealistic to claim that "this kind of bias seems to be actually desirable for large specifications, because it helps the implementer in his difficult task of structuring the overall program design." Indeed, this may well be the case sometimes, but it is certainly not always true.

As presented at length in Sect. 5, we are very far from the view that horizontal composability is unimportant. However, we believe that one should carefully distinguish and keep separate two conceptually different roles that the horizontal structure of a specification may play. One is the usual structuring of specifications, used to present the concepts of the problem space in a clear and perspicuous way. The horizontal structure obtained in this way is in principle irrelevant for vertical development, although it may be used when appropriate. The other role is the design of the modular structure of the system to be developed. This may be viewed as a very special kind of horizontal structure, which indeed is required to be preserved throughout development. Horizontal composability with respect to *this* structure is crucial, of course, and the double law is a natural and useful consequence. We proposed architectural specifications as a tool for capturing horizontal structure of this latter kind. We feel that the overall picture of vertical development and its interplay with this horizontal structure, as imposed by architectural specifications and sketched in Sects. 4 and 5, give a well-founded account of the ideas that were put forward in [GB80].

Acknowledgements: Hearty congratulations to Joseph on his 65th birthday and our thanks to him for the many novel ideas that over the years have stimulated much of our own work 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.

[BCH+04] H. Baumeister, M. Cerioli, A. Haxthausen, T. Mossakowski, P.D. Mosses, D. Sannella, and A. Tarlecki. CASL semantics. [CoF04], part III, pages 115–273. D. Sannella and A. Tarlecki, editors.

[BG80] R.M. Burstall and J.A. Goguen. The semantics of CLEAR, a specification language. In *Proceedings of the Abstract Software Specifications, 1979 Copenhagen Winter School*, Springer LNCS 86, pages 292–332, 1980.

[BM04] M. Bidoit and P.D. Mosses. CASL *User Manual*. Springer LNCS 2900 (IFIP Series). 2004. With chapters by T. Mossakowski, D. Sannella, and A. Tarlecki.

[Bor02] T. Borzyszkowski. Logical systems for structured specifications. *Theoretical Computer Science*, 286:197–245, 2002.

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

[Bur84] R.M. Burstall. Programming with modules as typed functional programming. In *Proc. Intl. Conference on Fifth Generation Computing Systems, Tokyo*, pages 103–112, 1984.

[CoF04] CoFI (The Common Framework Initiative). CASL *Reference Manual*. Springer LNCS 2960 (IFIP Series). 2004.

[Ehr82] H.-D. Ehrich. On the theory of specification, implementation and parameterization of abstract data types. *Journal of the Association for Computing Machinery*, 29:206–227, 1982.

[EKMP82] H. Ehrig, H.-J. Kreowski, B. Mahr, and P. Padawitz. Algebraic implementation of abstract data types. *Theoretical Computer Science*, 20:209–263, 1982.

[FJ90] J. Fitzgerald and C.B. Jones. Modularizing the formal description of a database system. In *Proc. 3rd Intl. Symp. VDM Europe: VDM and Z, Formal Methods in Software Development*, Springer LNCS 428, pages 189–210, 1990.

[GB80] J.A. Goguen and R.M. Burstall. CAT, a system for the structured elaboration of correct programs from structured specifications. Technical Report CSL-118, Computer Science Laboratory, SRI International, 1980.

[GB92] J.A. Goguen and R.M. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, January 1992. An early version appeared under the title "Introducing Institutions" in *Logics of Programs*, Springer LNCS 164, 221–256, 1984.

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

[GT79] J.A. Goguen and J. Tardo. An introduction to OBJ: A language for writing and testing software specifications. In M. K. Zelkowitz, editor, *Specification of Reliable Software*, pages 170–189. IEEE Press, Cambridge (MA, USA), 1979. Reprinted in *Software Specification Techniques*, N. Gehani and A. McGettrick, editors, Addison-Wesley, 1985, pages 391–420.

[GT00] J.A. Goguen and W. Tracz. An implementation-oriented semantics for module composition. In *Foundations of Component-Based Systems*, pages 231–263. Cambridge University Press, 2000. Edited by G. Leavens and M. Sitaraman.

[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. 1978. Edited by R.T. Yeh.

[Hoa72] C.A.R. Hoare. Correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

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

[KST97] S. Kahrs, D. Sannella, and A. Tarlecki. The definition of Extended ML: A gentle introduction. *Theoretical Computer Science*, 173:445–484, 1997.

[LF97] A. Lopes and J. Fiadeiro. Preservation and reflection in specification. In *Proc. 6th Intl. Conference on Algebraic Methodology and Software Technology, AMAST 1997*, Springer LNCS 1349, pages 380–394, 1997.

[LPRS88] P. Lee, F. Pfenning, G. Rollins, and W. Scherlis. The Ergo support system: An integrated set of tools for prototyping integrated environments. In *Proc. 3rd ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 25–34, 1988.

[Mac71] S. MacLane. *Categories for the Working Mathematician.* Springer, 1971.

[MHAH04] T. Mossakowski, P. Hoffman, S. Autexier, and D. Hutter. CASL logic. [CoF04], part IV, pages 275–361. T. Mossakowski, editor.

[Mil71] R. Milner. An algebraic definition of simulation between programs. In *Proc. 2nd Intl. Joint Conf. on Artificial Intelligence*, pages 481–489, 1971.

[MST04] T. Mossakowski, D. Sannella, and A. Tarlecki. A simple refinement language for CASL. In *Recent Trends in Algebraic Development Techniques: Selected Papers from WADT 2004*, Springer LNCS 3423, pages 162–185, 2004.

[MTHM97] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[SS83] W. Scherlis and D. Scott. First steps towards inferential programming. In *IFIP Congress*, pages 199–212, 1983.

[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.

[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.

[ST89] D. Sannella and A. Tarlecki. Toward formal development of ML programs: Foundations and methodology. In *Proc. Colloq. on Current Issues in Programming Languages. Intl. Joint Conf. on Theory and Practice of Software Development (TAPSOFT'89)*, Springer LNCS 352, pages 375–389, 1989.

[ST97] D. Sannella and A. Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Computing*, 9:229–269, 1997.

[ST99] D. Sannella and A. Tarlecki. Algebraic preliminaries. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specification*, chapter 2. Springer, 1999.

[SW83] D. Sannella and M. Wirsing. A kernel language for algebraic specification and implementation. In *Proc. 1983 Intl. Conf. on Foundations of Computation Theory*, Springer LNCS 158, pages 413–427, 1983.