Toward Formal Development of Programs from Algebraic Specifications: Model–Theoretic Foundations[1]

Donald Sannella
Department of Computer Science
University of Edinburgh
Edinburgh, UK

Andrzej Tarlecki
Institute of Computer Science
Polish Academy of Sciences
Warsaw, Poland

Abstract: This paper presents in an informal way the main ideas underlying our work on the model-theoretic foundations of algebraic specification and program development. We attempt to offer an overall view, rather than new results, and focus on the basic motivation behind the technicalities presented elsewhere and on the conclusions from this work.

Introduction

The long-term goal of work on algebraic specification is to provide a formal basis to support the systematic development of correct programs from specifications by means of verified refinement steps. There has been a large body of technical work directed towards this important goal. Many interesting technical concepts have been introduced and quite a number of non-trivial results have been stated and proved (see [BKLOS 91] for a review and a comprehensive list of references). Instead of providing yet another piece in the puzzle, in this paper we try to sketch on a rather informal level our views on how some of the existing pieces fit into an overall picture of what is important in the light of the ultimate goal. We focus on the motivations for certain technicalities which we think are of crucial importance, and try to draw some conclusions.

Our earlier papers already mentioned many of the points we make here, such as the use of "institutions" to ensure sufficient generality of the proposed framework, and the use of "constructor implementations" to capture the essence of program development steps (including "design" steps which involve a decomposition into independent programming tasks). Some of these ideas were hidden amongst the technical definitions and results, and so we think they are worth restating here more prominently, with more careful arguments in some cases. For example, we go into a bit more detail to justify our conviction that model-classes, not theories, form the appropriate semantic domain for specifications. We also present more clearly our current views on the role of behavioural equivalence in the development process.

Since the emphasis here is on motivation and intuition, we refer the interested reader to the papers we have published on these and related topics over the last few years and to a forthcoming monograph [ST 93] for the corresponding technical details.

[1]To appear in *Proc. 19th Intl. Colloq. on Automata, Languages and Programming*, Vienna. Springer LNCS (1992).

1 The logical framework

The overall aim of work on algebraic specification is to provide semantic foundations for the development of programs that are *correct* w.r.t. to their requirements specifications. In other words, the program developed must exhibit the required input/output behaviour. We view the correctness of a program as its most crucial property. Other desirable properties (efficiency, robustness, reliability etc.) are disregarded in this work. Of course, this does not mean that we do not care about them, but this approach does not provide any formal means for their analysis.

The assumption that the correctness of the input/output behaviour of a program takes precedence over all its other properties allows us to abstract away from concrete details of code and algorithms, and to model program functions as mathematical functions. Such functions are never considered in isolation, but always in units (program modules) comprising a collection of related functions together with the data domains they operate on. At this level of abstraction we are dealing directly with the information essential for the analysis of program correctness, without the burden of irrelevant details. This leads to the most fundamental assumption underlying work on algebraic specification: programs are modelled as many-sorted *algebras*.

We refrain from recalling the formal definition of many-sorted algebra (see e.g. [EM 85]). It is enough to know that an algebra consists of a collection of *carriers* (sets of data) and *operations* on them. Algebras are classified by *signatures*, naming the algebra components (sorts and operations) and thus providing the basic vocabulary for using the program and for making assertions about its properties. The class of all Σ-algebras (algebras over the signature Σ) will be denoted by $Alg(\Sigma)$. For any program P, the algebra it denotes is written as $[\![P]\!] \in Alg(Sig(P))$, where $Sig(P)$ is the underlying signature of P.

For any signature, we need a logical system for describing properties of algebras over that signature. Many-sorted equational logic (cf. [GM 85, EM 85]) is the most commonly-used system for this purpose, at least in the area of algebraic specification. Properties of Σ-algebras (or rather, of their operations) may be described by *equations* over Σ: we know what it means for a Σ-algebra A to *satisfy* a Σ-equation φ, written $A \models \varphi$. This also determines a notion of *logical consequence*: a set of axioms Φ entails an equation φ, written $\Phi \models \varphi$, if every algebra that satisfies all the axioms in Φ also satisfies φ.

Very rarely in the process of program development does the user work with just a single signature: operations and sorts of data are renamed, added and hidden according to what seems most suitable. To take account of this, signatures are equipped with a notion of *signature morphism* (cf. [EM 85]). A signature morphism $\sigma : \Sigma \rightarrow \Sigma'$ maps the sort and operation names of Σ to those of Σ'. This determines in a natural way a translation of any Σ-equation φ to a Σ'-equation $\sigma(\varphi)$, and on the semantic level, a translation of any Σ'-algebra $A' \in Alg(\Sigma')$ to its *reduct* $A'|_\sigma \in Alg(\Sigma)$. (Notice the change of direction!)

The above framework is often criticised (quite rightly!) as rather restrictive and cumbersome to use in practice. Some important features of programs, for example non-termination and higher-order functions, are difficult to model in algebras; equations are often not expressive enough to conveniently capture properties which one may want to state as requirements. Fortunately, this deficiency is relatively easy to overcome using the concept of *institution*. This concept was introduced by Goguen and Burstall [GB 84] to capture the informal notion of a logical system and was strongly influenced by the understanding of this notion in the theory of specifications (see [Bar 74] for an early account of *abstract model theory* covering similar ideas approached from the viewpoint of classical logic and model theory).

An institution defines a notion of a signature together with for any signature Σ, a set of Σ-sentences, a class of Σ-models and a satisfaction relation between Σ-models and Σ-sentences. Moreover, signatures come equipped with a notion of a signature morphism. Any signature morphism induces a translation of sentences and a translation of models (the latter going into the opposite direction as above). The only semantic requirement is that when we change signatures using a signature morphism, the induced translations of sentences and of models preserve the satisfaction relation. Many standard logical systems have been presented explicitly as institutions (cf. [GB 90]). It should be easy to see that in fact any usual logical system with a well-defined model theory may be put into this mould.

Everything below, barring concrete examples, works in the framework of an arbitrary institution, even though for the reader's convenience we avoid "institutional jargon" and use more familiar terms (signature, algebra, axiom). Consequently, everything in this paper applies to all of the many different concepts of "signature", "algebra" and "axiom" used in the theory and practice of software specification. This is a general feature of our line of research: most of the technicalities and ideas are parameterised by an arbitrary institution. This results in "reusable" methodologies, theorems, and (ultimately) tools. This approach originated in [GB 84]; in [ST 87, ST 88a, ST 88b, ST 91a] various components of a framework for software specification and development have been elaborated following this idea.

Strict followers of the early approaches to algebraic specification might view this as an alarming departure, and might protest that what we are doing is not algebraic specification at all. In our view the essential idea of algebraic specification is the stress on "algebra-like" models and the use of logical axioms to describe such models. The use of ordinary many-sorted algebras and equations is but a special case of this. Just as it was necessary to generalise from classical single-sorted algebras to many-sorted algebras in order to deal with programs handling several kinds of data, it is necessary to adopt more complicated models to deal with other features of programming languages (polymorphism, higher-order functions, infinite behaviour, lazy evaluation, etc.). The essence is that we need a notion of a semantic structure which is detailed enough to capture the program properties we want to analyse and abstract enough to make this analysis feasible. Moreover, to specify and reason about programs, we need a logical system with a model theory based on such structures.

2 Specifications

What is a specification? Clearly, since our aim is a formal approach to software development, specifications must be objects as formal as (for example) programs are. That is, we have to have a formal language to write specifications down and to provide a vehicle on which formal techniques to manipulate specifications may be based. It is important for such a *specification language* to provide a collection of convenient notational conventions which are easy to understand and use. One of the basic constituents of a specification will be a list of axioms the specified program is required to satisfy.

Any specification language must be given a precise, formal semantics. Here, the first question to ask is what the meanings of specifications are, i.e. what specifications denote. Whatever the full answer is, a specification at least determines the underlying signature of the specified system. For any specification SP, we write this signature as $Sig(SP)$. Then, one may attempt to give a semantics of specifications on (at least) three different levels:

- Presentation level: a specification denotes a signature and a set of axioms over this signature (this set may be required to be finite or at least recursive or recursively enumerable). At this level, the meaning of a specification is close to the syntactic form in which specifications are written; the semantics extracts the axioms, resolves references to other specifications, etc.

- Theory level: a specification denotes a signature and a set of axioms over this signature which is closed under logical consequence. At this level, the set of axioms is much larger than what has been written explicitly (it is typically infinite, usually not recursive and sometimes not r.e.); thus the meaning of a specification is no longer strictly syntactic. The semantics performs the closure under logical consequence.

- Model-class level: a specification denotes a signature and a class of algebras over this signature. At this level, the meaning of a specification is entirely non-syntactic (except for the signature part). The semantics abstracts away from the axioms, taking into account only their possible realizations.

The ultimate role of any specification is to describe a class of programs which we want to view as its correct realizations. Since we have already decided to model programs as algebras, whichever one of these three levels we choose for the semantic domain, given the natural mappings from presentations to theories and from theories to model classes, *ultimately specifications determine classes of algebras.*

For any specification SP, the algebras that model programs which are considered to be correct realizations of SP are referred to as the *models* of SP, and the class of all models of SP is denoted by $[\![SP]\!] \subseteq Alg(Sig(SP))$. This semantics determines a notion of logical consequence of a specification: a specification SP entails an axiom φ, written $SP \models \varphi$, if φ holds in every model of SP.

Of course, a specification SP may admit a number of different program behaviours, and so the class $[\![SP]\!]$ may contain non-isomorphic algebras (hence we cover so-called *loose* specifications). Or it might be empty.

For any signature Σ, there is a Galois connection between classes of Σ-algebras and sets of Σ-axioms, assigning to any set of axioms the class of all algebras that satisfy them, and to any class of algebras the set of all axioms that hold in them (see [GB 84]). The closed elements of this Galois connection are theories, that is, sets of axioms determined by classes of algebras; they are in one-to-one correspondence with closed (i.e., definable by sets of axioms) classes of algebras. Therefore, it is obvious that as a semantic domain for specifications the theory level is less expressive than either the presentation or the model-class level. The latter two are, however, incomparable: there are properties which can be naturally studied at the presentation level (for example, finiteness of an axiomatisation) with no natural counterpart at the model-class level, and vice versa.

It is not immediately obvious that working at the model-class level brings any essential benefits over working with closed classes of algebras only, or equivalently, working at the theory level. It is not clear whether non-closed classes of algebras ever arise as meanings of specifications; even if they do arise, it is not clear whether this makes any difference for the use of specifications. The following example exhibits both of these phenomena (we use some *ad hoc* notation for writing specifications with a hopefully clear intuitive interpretation):

$$SP \left\{ \begin{array}{l} \textbf{enrich} \\ \qquad SP_1 \left\{ \begin{array}{l} \textbf{hide } a \textbf{ in} \\ \qquad SP_0 \left\{ \begin{array}{ll} \textbf{sorts} & s, s' \\ \textbf{opns} & a : s \\ & b, c : s' \end{array} \right. \end{array} \right. \\ \textbf{by} \quad \forall x{:}s.\, b = c \end{array} \right.$$

This example relies on the following well-known phenomenon [GM 85]: $\forall x{:}s.\, b = c$ does not imply $b = c$, although it implies $b = c$ for $Sig(SP)$-algebras with non-empty carrier of sort s.

Now, at the model-class level, $[\![SP_0]\!]$ is the class of all algebras (over the indicated signature) and $[\![SP_1]\!]$ consists of all algebras that are reducts of $Sig(SP_0)$-algebras, obtained by forgetting the constant a. Consequently, $[\![SP_1]\!]$ contains only those algebras having a non-empty carrier of sort s. Then, selecting the algebras that satisfy $\forall x{:}s.\, b = c$ yields the class $[\![SP]\!]$ — and all these algebras satisfy $b = c$ (since for the algebras in $[\![SP_1]\!]$, $b = c$ follows from $\forall x{:}s.\, b = c$). Thus, under the model-class interpretation, the property $b = c$ is a consequence of the specification SP.

On the other hand, at the theory level, the theory of SP_0 is clearly the trivial equational theory, and so is the theory of SP_1 (there are no equations capable of expressing the fact that a carrier is non-empty). Then, the additional axiom $\forall x{:}s.\, b = c$ in the context of the theory of SP_1 does not entail the equation $b = c$. Thus, under the theory-level interpretation, $b = c$ is *not* ensured by the specification SP.

This discrepancy (and similar examples one may construct without relying on the "empty carriers" phenomenon) faces us with the necessity to choose between theories and classes of algebras as the basic semantic domain for specifications. The choice is obvious: the objects of ultimate interest here are programs, which are modelled as algebras, while axioms and theories are nothing more than logical means for describing them. The lack of agreement between theories and classes of algebras shows that theories are *not* adequate as denotations of specifications.

The above specification SP is a trivial example of a *structured* specification, in which a complex specification is built from simpler ones. Any specification formalism must offer such a possibility if it is to be used in practice: a specification of a real-life system must state a huge number of properties, and building such a specification in an unstructured, monolithic way would result in a long list of axioms which would be neither understandable nor useful. Moreover, the structure of a specification may be used to express intangible aspects of the specifier's knowledge of the problem. For this purpose, a specification language must provide some *specification-building operations* used to put together small specifications to form more complex ones [BG 77]. Then, an understanding of a large specification is achieved via an understanding of its components. Since we have chosen classes of algebras as meanings of specifications, such specification-building operations semantically correspond simply to functions on classes of algebras.

Choosing appropriate specification-building operations to be included in a specification language is a non-trivial task (even though by now there is a list of typical ones that specification languages are based on). It involves a certain trade-off between the expressive power of the specification language and the ease of understanding and dealing with the operations. One way to circumvent this problem is to first develop a *kernel* language consisting of a minimal set of very powerful, but perhaps difficult to use operations, and then build on top of it a higher-level, more user-friendly language, perhaps sacrificing some of the expressive power to achieve ease of use and ease of understanding. Such an approach has been taken with the ASL [SW 83, ST 88a, Wir 86] kernel specification language, on top of which such specification languages as PLUSS [Gau 84] and Extended ML [ST 91b] have been built.

We should stress here that the internal structure of a specification should not constrain the final structure of its implementations. This is one of the consequences of the famous dogma that a specification should describe only the *what's* of the specified software without constraining any of its *how's*. In fact, requiring the structure of the initial specification to be preserved in its implementation would be highly unrealistic and unreasonable, even though this has been explicitly suggested by some (e.g. [GB 80, Mor 90]) and is implicit in the approach taken by others. The aims of structuring requirements specifications are often contradictory with the aims of structuring software. See for instance [FJ 90] for a nice discussion of a practical example where such a discrepancy occurs.

3 Specification engineering

The point of constructing a specification is so that it may be used to define a programming task by precisely delimiting the range of program behaviours that are to be regarded as permissible. This statement of the programming task represents a rather idealized view that the requirements specification we start with accurately reflects the real-life requirements the customer expects to be satisfied. Thus formulated, the programming task allows for no change to the original specification. Of course, this is not very realistic, and we envisage that a programmer working with an appropriate support system will be able to try slightly different ideas, to modify the original formal specification, to experiment with a whole bunch (or better, a tree) of specifications.

The problem of getting the original requirements specification right is the topic of "requirements engineering" [Par 91]. Even though there has been much work on this, we are not convinced that all the potential of the use of formal specifications has been explored. Naturally, there always will be a gap between the informal wishes of the customer and their formalised version couched as a precise, unambiguous, entirely formal specification. Some methods involved in the process of passing from the former to the latter may and should find a more precise formulation.

One major problem within this area is how we can make formal specifications easier to understand and construct. We have already mentioned a part of the answer in the previous section: we need good specification languages with good structuring operations allowing specifications to be built and understood in a systematic, modular fashion.

Another aspect here is that once a specification is built, the customer should be able to "play with" it, to test whether or not the specification indeed expresses the properties he expects. A traditional approach to this is to engage a team of programmers to build a *prototype*, a quickly assembled but necessarily bad and simplified realization. This can then be given to the customer to test. Of course, such an approach is irreplacable for some aspect of the software to be developed. For example, there can be no better way to test a user interface than by playing with some version of it; going through sample sessions with the system seems to be the only way for a user to get a feel for what working with the system will be like. In general, however, the prototyping approach has a number of disadvantages. First, it involves some extra work to produce a system which is then thrown away. More importantly, if the original specification is loose (and it usually is) any prototype will incorporate choices between the alternative behaviours permitted by the specification, and these choices need not necessarily be mirrored in the final implementation. Consequently, the user may conclude that the system will have some properties which are not ensured by the specification at all, and this undermines the sense of the whole exercise.

The overhead of prototyping may be avoided through the use of a rapid prototyping system like RAP [Hus 85]. This demands that requirements specifications be written in an executable specification language, not far from high-level programming languages like Standard ML [MTH 90]. In the fundamental trade-off between executability and expressiveness, it is clearly the latter which is of central importance in a language intended for writing requirements specifications, so such a strong restriction seems highly undesirable.

We believe that for many purposes prototyping should be replaced by *theorem proving* (see [GH 80] for a similar observation). To check whether a given specification indeed embodies a desirable property, it seems most appropriate to state this property explicitly and then try to prove that it is a consequence of the specification. This is the most general form of specification testing activity; the more usual approaches via rapid prototyping, symbolic evaluation, term rewriting etc. can easily be seen as some special cases, or rather as some special techniques of theorem proving applicable to some particular situations.

This indicates a need for good theorem provers. For use in this area, theorem provers should be able not only to derive consequences of a list of axioms ($\Phi \models \varphi$), but also to derive consequences of a specification built in a structured way ($SP \models \varphi$). Theorem provers should be able to exploit the structure of specifications to guide proof search (cf. [SB 83, ST 88a, HST 89, Wir 92]). It would also be extremely useful for a theorem prover, in the case where it fails to find a proof, to provide the user with readable information on where the proof attempts break down, and perhaps even how the specification may be augmented to make the proof go through — a desirable feature which few contemporary theorem provers exhibit.

4 Program development

Given a specification SP, the programming task it defines is to construct a program P which is a correct realization of SP, that is such that $[\![P]\!] \in [\![\![SP]\!]\!]$.

There can be no universal recipe which would ensure successful development of a program implementing a given specification. All we can hope to offer are methodologies, and perhaps some particular techniques and heuristics oriented towards specific problem areas.

Perhaps the most fundamental point is that it is not possible to leap in a single bound over the gap between a high-level user-oriented requirements specification and the very specific realm of programs full of technical decisions and algorithmic details. Program development should proceed systematically in a stepwise fashion, gradually enriching the original requirements specification with more and more detail, incorporating more and more design and implementation decisions. Such decisions include choosing between the options of behaviour left open by the specification, between the algorithms which realize this behaviour, between data representation schemes, etc. Each such decision should be recorded separately, as a separate step hopefully consisting of a local modification to the specification. The program development process is then a sequence of such small, easy to understand and easy to verify steps:

$$SP_0 \rightsquigarrow SP_1 \rightsquigarrow \cdots \rightsquigarrow SP_n$$

In such a chain, SP_0 is the original requirements specification and $SP_{i-1} \rightsquigarrow SP_i$ for any $i = 1, \ldots, n$ is an individual *refinement* step. The aim is to reach a specification (here, SP_n) which is an exact description of a program in full detail, with all the technical decisions incorporated (it may simply *be* a program, if our specification formalism is rich enough).

Any formal definition of such refinement steps $SP \rightsquigarrow SP'$ must incorporate the requirement that any correct final realization of SP' must be (or, somewhat more generally, must give rise to) a correct realization of SP. Recalling that $[\![SP]\!]$ is the class of all admissible realizations of SP, this leads to the following straightforward definition [SW 83, ST 88b]:

$$SP \rightsquigarrow SP' \quad \textbf{iff} \quad [\![SP']\!] \subseteq [\![SP]\!]$$

This definition ensures that the correctness of the final outcome of the stepwise development process may be inferred from the correctness of the individual refinement steps:

$$\frac{SP_0 \rightsquigarrow SP_1 \rightsquigarrow \cdots \rightsquigarrow SP_n \qquad A \in [\![SP_n]\!]}{A \in [\![SP_0]\!]}$$

The proof is by an easy induction on the length of the refinement sequence.

Notice that if the final specification SP_n represents an individual program P, then the conclusion that $A \in [\![SP_0]\!]$ for all $A \in [\![SP_n]\!]$ is equivalent to our original statement of the program development task: $[\![P]\!] \in [\![SP_0]\!]$.

An indirect way to prove the correctness of the final outcome is to notice a stronger fact, namely that consecutive refinements can be composed (referred to as "vertical composability" [GB 80]):

$$\frac{SP \rightsquigarrow SP' \qquad SP' \rightsquigarrow SP''}{SP \rightsquigarrow SP''}$$

The above gives a formal view of the stepwise development methodology. As mentioned before, there can be no universal recipe for coming up with useful refinements of a given specification — necessarily, this is the place where the developer's invention is required. Once a refinement step is proposed, though, we should be able to prove it correct, that is, we should have some formalism for proving the inclusion between the corresponding model classes. Of course, this must incorporate a theorem prover for the underlying logic. A new need which arises here is that such a formalism must also be able to prove entailments between two structured specifications (we write $SP' \models SP$ to state that every model of SP' is a model of SP, yet another formulation of $SP \rightsquigarrow SP'$ which we will use in this context). If the structures of SP and SP' match exactly (and the specification-building operations used are monotonic w.r.t. inclusion of model classes — this is typically the case) then this problem may be reduced to proving that individual axioms (from SP) are consequences of certain specifications (parts of SP') via the following fact (referred to as "horizontal composability" [GB 80] for the specification-building operation *op*):

$$\frac{SP_1 \rightsquigarrow SP'_1 \qquad \cdots \qquad SP_n \rightsquigarrow SP'_n}{op(SP_1, \ldots, SP_n) \rightsquigarrow op(SP'_1, \ldots, SP'_n)}$$

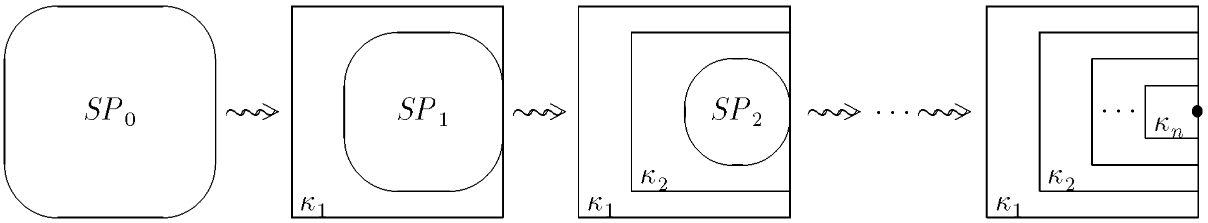
Unfortunately, the structures of the two specifications need not coincide, which makes such a reduction very non-trivial. The only work on this important problem we are aware of is [Far 92].

Another issue which may seem worrying here is that we have not put into our definition of refinement any requirement that the refined specification is consistent (that it has any model). Indeed, this can be seen as a problem, since an inconsistent specification cannot be implemented by any program, and so it opens a blind valley in the program development process. From this point of view, it would be worthwhile to be able to check consistency of a specification as soon as it is formulated. Unfortunately, in general (for any sufficiently powerful specification framework) this is an undecidable property. Fortunately, inconsistency of specifications cannot lead to incorrect programs: if we arrive at a program at some point in the development process, then this program is by definition consistent (it has a unique model) and consequently, all the specifications leading to it must have been consistent as well.

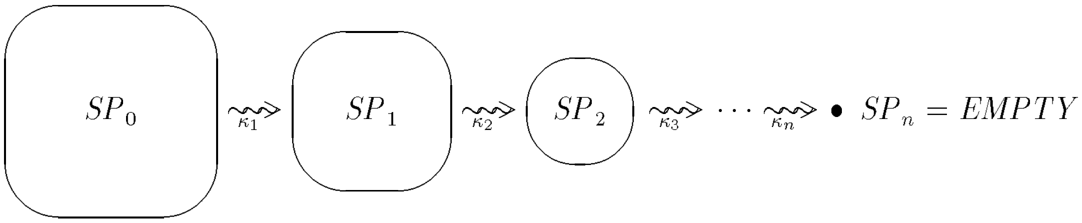
The proposed methodology of stepwise refinement does not and cannot be expected to guarantee success. Apart from inconsistencies, there are many sources of "blind valleys" and failures in the development process: there might be no computable realization of a specification, there might be no "practically computable" realization, we might not be clever enough to find a realization, we might run out of money to finish the project, etc. The main feature of the methodology we really can ensure is its *safety*: if we arrive at a program, then it is a correct realization of the original specification.

5 Constructor implementations

The simple notion of specification refinement is mathematically elegant and powerful enough (in the context of a sufficiently rich specification language) to handle all concrete examples of interest. However, it is not very convenient to use: in the practice of software development, certain constructions are used so often that it seems tempting to incorporate some treatment of them in the notion of refinement. 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.



It is more convenient to avoid such clutter by putting the finished parts aside, and proceeding with the development of the unresolved parts only.



It is important for these finished parts to be independent of the particular choice of realization for what is left: they should act as constructions extending any realization of the unresolved part to a realization of what is being refined. Semantically, each κ_i amounts to a function (which we will call a *constructor*) on algebras, $\kappa_i : Alg(Sig(SP_i)) \to Alg(Sig(SP_{i-1}))$. Once the development is finally finished (that is, when nothing is left unresolved) we can put the constructors together to obtain a correct realization of the original specification.

To formally capture the above considerations, we introduce the concept of *constructor implementation* [ST 88b], a more elaborate version of the notion of refinement of the previous section. We write $SP \underset{\kappa}{\leadsto\!\!>} SP'$ to say that a specification SP' implements a specification SP via a constructor $\kappa : Alg(Sig(SP')) \to Alg(Sig(SP))$, and define this as follows:

$$SP \underset{\kappa}{\leadsto\!\!>} SP' \quad \textbf{iff} \quad \kappa([\![SP']\!]) \subseteq [\![SP]\!]$$

Here, $\kappa([\![SP']\!])$ is the image of $[\![SP']\!]$ under κ. This suggests how a constructor κ may be viewed as a specification-building operation $\overline{\kappa}$, so that constructor implementations may be viewed as refinements ($SP \underset{\kappa}{\leadsto\!\!>} SP'$ is equivalent to $SP \leadsto\!\!> \overline{\kappa}(SP')$). Then proof techniques for refinements may be applied to establish the correctness of constructor implementations.

The correctness of the final outcome of the stepwise development process may be inferred from the correctness of the individual constructor implementation steps:

$$\frac{SP_0 \underset{\kappa_1}{\leadsto\!\!>} SP_1 \underset{\kappa_2}{\leadsto\!\!>} \cdots \underset{\kappa_n}{\leadsto\!\!>} SP_n = EMPTY}{\kappa_1(\kappa_2(\ldots\kappa_n(\langle\,\rangle)\ldots)) \in [\![SP_0]\!]}$$

where $EMPTY$ is the empty specification over the empty signature and $\langle\,\rangle$ is its unique (empty) model.

It is also easy to see that constructor implementations (vertically) compose:

$$\frac{SP \underset{\kappa}{\leadsto\!\!>} SP' \qquad SP' \underset{\kappa'}{\leadsto\!\!>} SP''}{SP \underset{\kappa';\kappa}{\leadsto\!\!>} SP''}$$

(semicolon stands for functions composition, written in the diagrammatic order). As in the case of refinement, vertical composability is not really necessary to ensure the correctness of the development process. All we need is the condition inherent in the definition of constructor implementation, namely that implementations reflect realizations:

$$\frac{SP \underset{\kappa}{\leadsto\!\!>} SP' \qquad A' \in [\![SP']\!]}{\kappa(A') \in [\![SP]\!]}$$

Even though in the above we have viewed constructors as arbitrary functions on algebras, in practice more restrictions should be imposed. In particular, we want constructors to be "effective": given a contructor implementation $SP \underset{\kappa}{\leadsto\!\!>} SP'$, the ability to compute in an algebra $A' \in [\![SP']\!]$ should ensure the ability to compute in the algebra $\kappa(A') \in [\![SP]\!]$. In this sense, κ may be viewed as a *parameterised program* [Gog 84] or, equivalently, as a Standard ML *functor* [MTH 90] with input interface SP' and output interface SP. A programming language which supports stepwise development in the style suggested here will provide syntax and modularisation facilities for coding up such constructors. The modularisation facilities must ensure that one may successively instantiate constructors in order to assemble the final program $\kappa_1(\kappa_2(\ldots\kappa_n(\langle\,\rangle)\ldots))$. This is a much weaker requirement than that constructors be composable (necessary for vertical composability of constructor implementations in the specific programming language) — even though any programming language with decent modularisation facilities should ensure the latter as well. It seems that this stronger requirement becomes important when higher-order parameterised programs and their development from specifications are considered [SST 90].

Much work on vertical composability of implementations (see e.g. [EKMP 82, SW 82, Ore 83]) has aimed at a still stronger requirement. This stems from the choice of a definition of implementation which is similar to that of constructor implementation but requires the constructor to be in a particular fixed form. Then the vertical composition of two implementations must yield an implementation of the same form. The requirement that the composition of constructors be forced into some given normal form corresponds to requiring programs to be written in a rather restricted programming language which does not provide sufficiently powerful modularisation facilities for the job.

We have already mentioned that the internal structure of a requirements specification need not be mirrored by programs which realize it. This is why the definitions of refinement and constructor implementation above take no account of the structure of specifications. However, when developing a large program it is crucial to progressively decompose the job into smaller tasks which can be handled separately. Each task is defined by a specification, and solving a task means producing a program component which satisfies this specification. Once all tasks are solved, then producing the final system is a matter of appropriately assembling these components.

A development step involving the decomposition of a programming task into separate subtasks is modelled using a constructor implementation with a multi-argument constructor:

$$SP \underset{\kappa}{\rightsquigarrow} \langle SP_1, \ldots, SP_n \rangle \quad \textbf{iff} \quad \kappa([\![SP_1]\!], \ldots, [\![SP_n]\!]) \subseteq [\![SP]\!]$$

where $\kappa : Alg(Sig(SP_1)) \times \cdots \times Alg(Sig(SP_n)) \to Alg(Sig(SP))$ is an n-argument constructor (an n-argument function on algebras) describing a way to put realizations of $SP_1, \ldots, SP_n$ together to construct a realization of SP. Now the development process takes on a tree-like shape. This process is finished once a tree is obtained which has empty (sequences of) specifications as its leaves:

$$SP \underset{\kappa}{\rightsquigarrow} \begin{cases} SP_1 \underset{\kappa_1}{\rightsquigarrow} \langle\,\rangle \\ \quad\vdots \\ SP_n \underset{\kappa_n}{\rightsquigarrow} \begin{cases} SP_{n1} \underset{\kappa_{n1}}{\rightsquigarrow} \left\{ SP_{n11} \underset{\kappa_{n11}}{\rightsquigarrow} \langle\,\rangle \right. \\ \quad\cdots \\ SP_{nm} \underset{\kappa_{nm}}{\rightsquigarrow} \langle\,\rangle \end{cases} \end{cases}$$

Then the composition of the constructors in the tree yields a realization of the original requirements specification. The above tree yields:

$$\kappa(\kappa_1(), \ldots, \kappa_n(\kappa_{n1}(\kappa_{n11}()), \ldots, \kappa_{nm}())) \in [\![SP]\!].$$

The structure of the final program is determined by the shape of the development tree, which is in turn determined by the decomposition steps. Each such step corresponds to what software engineers call a *design* specification: it defines the structure of the system by specifying its components and describing how they fit together. This style of development leads to modular programs, built from fully specified, correct and reusable components.

Horizontal composability for constructor implementations takes the form:

$$\frac{SP_1 \underset{\kappa_1}{\rightsquigarrow} SP'_1 \qquad \cdots \qquad SP_n \underset{\kappa_n}{\rightsquigarrow} SP'_n}{op(SP_1,\ldots,SP_n) \rightsquigarrow op(\overline{\kappa_1}(SP'_1),\ldots,\overline{\kappa_n}(SP'_n))}$$

In spite of this fact, which holds for all monotonic specification-building operations *op*, decomposition of a specification $SP = op(SP_1,\ldots,SP_n)$ into separate tasks $SP_1,\ldots,SP_n$ might not be appropriate. It is possible for the design decisions taken in the solutions of these separate tasks to conflict so that even once we have obtained realizations of $SP_1,\ldots,SP_n$, it might not be possible to combine these to form a realization of SP. This problem cannot arise for specification-building operations corresponding to constructors, as in the decomposition steps via multi-argument constructors above.

6 Behavioural implementations

A specification should be a precise and complete statement of required properties. We should try to avoid including extra requirements, even if they happen to be satisfied by a possible future realization. Such over-specification unnecessarily limits the options left open to the implementer. Ideally, the target is to describe exactly the admissible program behaviours. This suggests that specifications of programming tasks should not distinguish between programs (modelled as algebras) exhibiting the same behaviour.

The intuitive idea of *behaviour* of an algebra has been formalised in a number of ways (see e.g. [Rei 81, GM 82, SW 83, Sch 87, ST 87]). In most approaches one distinguishes a certain set *OBS* of sorts as *observable*. Intuitively, these are the sorts of data directly visible to the user (integers, booleans, characters, etc.) in contrast to sorts of "internal" data structures, which are observable only via the functions provided by the program. The behaviour of an algebra is characterised by the set of *observable computations* taking arguments of sorts in *OBS* and producing a result of a sort in *OBS*. Two Σ-algebras A and B are *behaviourally equivalent* (w.r.t. *OBS*), written $A \equiv B$, if they exhibit the same behaviour, that is, if all observable computations yield the same results in A and in B. The motivation is related to that of so-called *testing* equivalences studied in the context of concurrent systems [DH 84].

A hackneyed example is that of stacks of integers, with the usual operations (`empty`, `isempty`, `push`, `pop`, `top`). The sorts `int` and `bool` are observable while the sort `stack` is not. The observable computations are all the terms of the form `isempty(`s`)` and `top(`s`)` where s is a term of sort `stack` with variables (representing the inputs) of sort `int` only. Now, all intuitively acceptable realizations of stacks are behaviourally equivalent, since for any observable computation (like `top(pop(push(`n`,push(4,push(6,empty)))))`) they all deliver the same result (in this case `4`). However not all of these algebras act the same way when non-observable computations are considered: for example, the computations `empty` and `pop(push(`n`,empty))` yield the same result when stacks are represented as lists, but they yield different results when a stack is represented by an array with a pointer to the top element (the latter computation then leaves n in the position above the pointer).

Our earlier discussion would lead us to expect the class of models of a specification to be closed under behavioural equivalence. It is perhaps surprising that this is not easy to achieve directly: the class of models of a set of axioms typically does not have this property. Equational logic may be modified so as to force this to happen (cf. [NO 88]) but we see no straightforward way to achieve this for most logical systems. Instead, we suggest simply closing the class of models of a specification under behavioural equivalence [SW 83, ST 87]. Any specification SP determines the class $[\![SP]\!] \subseteq Sig(SP)$ of models which "literally" satisfy the stated requirements; the ultimate semantics of SP is taken to be the closure of this under behavioural equivalence:

$$[\![\![SP]\!]\!] = \{A \mid A \equiv B \text{ for some } B \in [\![SP]\!]\}$$

If SP is a list of axioms, then $[\![SP]\!]$ contains exactly the algebras which satisfy all the axioms, while $[\![\![SP]\!]\!]$ contains also the algebras which do not satisfy the axioms themselves but are behaviourally equivalent to algebras which do. For example, if the specification $STACK$ consists of the usual stack axioms, then both the list representation and the array-with-pointer representation of stacks are in $[\![\![STACK]\!]\!]$, even though the latter does not literally satisfy the axiom $\mathtt{pop(push(}n\mathtt{,}s\mathtt{))} = s$ and so is not in $[\![STACK]\!]$. This approach gives extra expressive power to the system: there are classes of algebras which may be finitely characterised in this way, and which cannot be finitely axiomatised directly [Sch 91]. Also, *model-oriented specifications* [Jon 86] can be handled: if $[\![SP]\!]$ contains just a single algebra, $[\![\![SP]\!]\!]$ admits any realisation of the exhibited behaviour.

The basic intuition for the use of behavioural equivalence in the development process is that it is not necessary to implement a specification SP according to its literal interpretation $[\![SP]\!]$; it is sufficient to implement it up to behavioural equivalence, as captured by its "ultimate" semantics $[\![\![SP]\!]\!]$. The crucial novelty, due to [Sch 87], is that when *using* a realization of SP, it is convenient (and possible) to pretend that it satisfies the literal interpretation of SP. For example, consider the following program:

```
multipush(n,s) = if n=0 then s else multipush(n-1,push(anything,s))
multipop(n,s) = if n=0 then s else multipop(n-1,pop(s))
id(x,n) = top(multipop(n,multipush(n,push(x,empty))))
```

Given any realization of $STACK$, to verify that $\mathtt{id(}x\mathtt{,}n\mathtt{)} = x$ for all x and $n \geq 0$, it is convenient to assume that the axiom $\mathtt{pop(push(}n\mathtt{,}s\mathtt{))} = s$ holds literally — then a simple proof by induction goes through — in spite of the fact that this equation is not valid in $[\![\![STACK]\!]\!]$. These considerations lead to the following definition of *behavioural implementation* [ST 88b]:

$$SP \underset{\kappa}{\overset{\equiv}{\leadsto}} SP' \quad \textbf{iff} \quad \kappa([\![\![SP']\!]\!]) \subseteq [\![\![SP]\!]\!]$$

The alert reader will have noticed that there is a problem here: we want to have our cake and eat it. On one hand, we want to allow specifications to be implemented up to behavioural equivalence; on the other hand, we would like to use any realization as if it satisfied its specification literally. Behavioural implementations do not compose, and the following crucial property is lost:

$$\frac{SP \underset{\kappa}{\overset{\equiv}{\leadsto}} SP' \qquad A' \in [\![\![SP']\!]\!]}{\kappa(A') \in [\![\![SP]\!]\!]}$$

The behavioural implementation $SP \underset{\kappa}{\overset{\equiv}{\leadsto}} SP'$ ensures only that algebras in $[\![SP']\!]$ give rise to correct realizations of SP; this says nothing about the models in $[\![\![SP']\!]\!]$ which are not in $[\![SP']\!]$.

It might seem that all is lost. But there is a way out, originally suggested in [Sch 87]. The above crucial property is recovered if we assume that the constructors used are *stable*, that is, that any constructor $\kappa : Alg(Sig(SP')) \to Alg(Sig(SP))$ preserves behavioural equivalence:

$$\text{Stability assumption:} \qquad \text{if } A \equiv B \text{ then } \kappa(A) \equiv \kappa(B)$$

(the exact definition of stability of constructors in a formal development framework based on a full-blown programming language is somewhat more complex — see [Sch 87, ST 89]).

Under this assumption, the correctness of the individual implementation steps ensures the correctness of the result:

$$\frac{SP_0 \underset{\kappa_1}{\overset{\equiv}{\leadsto\!\!>}} SP_1 \underset{\kappa_2}{\overset{\equiv}{\leadsto\!\!>}} \cdots \underset{\kappa_n}{\overset{\equiv}{\leadsto\!\!>}} SP_n = EMPTY}{\kappa_1(\kappa_2(\ldots\kappa_n(\langle\rangle)\ldots)) \in [\![SP_0]\!]}$$

We could repeat here the tree-like development picture of Section 5 — developments involving decomposition steps based on behavioural implementations with multi-argument (stable) constructors yield correct programs as well. We also recover vertical composability:

$$\frac{SP \underset{\kappa}{\overset{\equiv}{\leadsto\!\!>}} SP' \qquad SP' \underset{\kappa'}{\overset{\equiv}{\leadsto\!\!>}} SP''}{SP \underset{\kappa';\kappa}{\overset{\equiv}{\leadsto\!\!>}} SP''}$$

The correctness of a behavioural implementation $SP \underset{\kappa}{\overset{\equiv}{\leadsto\!\!>}} SP'$ is easier to verify than the correctness of the corresponding constructor implementation $SP \underset{\kappa}{\leadsto\!\!>} SP'$ ($\kappa([\![SP']\!]) \subseteq [\![SP]\!]$ is weaker than $\kappa([\![SP']\!]) \subseteq [\![SP]\!]$). We are still left, though, with the need to establish the stability of constructors, and so one may wonder if it is worthwhile taking advantage of this. However, the important point is that the constructors which may be used in program development are determined by the particular programming language to be used. Thus stability can be checked in advance, for the programming language as a whole (this is simplified somewhat by the fact that the composition of stable constructors is stable) and this frees the programmer from the need to prove it during the program development process.

There is a close connection between the requirement of stability and the security of encapsulation mechanisms in programming languages supporting abstract data types. A programming language ensures stability if the only way to access an encapsulated data type is via the operations explicitly provided in its output interface. This suggests that stability of constructors is an appropriate thing to expect; following [Sch 87] we view the stability requirement as a methodologically justified design criterion for the modularisation facilities of programming languages.

7 Final remarks

We have outlined the main ideas of a framework to support the formal development of correct programs from specifications. One of the central themes we did not have space to discuss is the vital role of parameterisation in specification and formal development. Following the technicalities in [SST 90], the framework outlined here may be extended to deal with the development of parameterised programs (modelled as functions on algebras) from their specifications. Extra flexibility is obtained by allowing higher-order parameterisation. Specifications of such parameterised programs (denoting classes of functions on algebras) should be carefully distinguished from parameterised specifications (denoting functions on classes of algebras).

Principle among the areas in which further work is required is the development of adequate proof technology. As mentioned earlier, much more than methods for proving consequences of unstructured sets of assumptions is required: we need to prove consequences of structured specifications, entailment between structured specifications, and correctness of behavioural implementation steps. Soundness with respect to the underlying model-theoretic semantics of specifications and of implementation steps is essential; completeness is unfortunately not achievable. In spite of some recent work, it is not clear how to scale up the methods and techniques which exist for the unstructured case (e.g. in the area of term rewriting) to deal with large structured specifications and with behavioural equivalence.

The main challenge now is to put these ideas into practice in the formal development of non-trivial programs in real programming languages. The above comments concerning proof technology indicate just a part of what is required. We are moving in this direction with our work on the Extended ML framework for the formal development of modular Standard ML programs [ST 91b], though more effort is required. Subjecting foundational work to the test of practice is sure to bring fascinating new problems and issues to light.

Acknowledgements: Most of the above ideas have been presented elsewhere, and have been discussed with and influenced by many of our colleagues. Thanks especially to Rod Burstall, Jordi Farrés, Joseph Goguen, Fernando Orejas, Oliver Schoett and Martin Wirsing. Partial support for writing this paper was provided by the ESPRIT-funded COMPASS working group and by a grant from the UK Science and Engineering Research Council.

References

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

[BKLOS 91] M. Bidoit *et al* (eds.) *Algebraic System Specification and Development: A Survey and Annotated Bibliography*. Springer LNCS 501 (1991).

[BG 77] R. Burstall and J. Goguen. Putting theories together to make specifications. *Proc. 5th Intl. Joint Conf. on Artificial Intelligence*, Cambridge, Massachusetts, 1045–1058 (1977).

[DH 84] R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science* 34, 83–133 (1984).

[EKMP 82] H. Ehrig, H.-J. Kreowski, B. Mahr and P. Padawitz. Algebraic implementation of abstract data types. *Theoretical Computer Science* 20, 209–263 (1982).

[EM 85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification I: Equations and Initial Semantics*. Springer (1985).

[Far 92] J. Farrés-Casals. Verification in ASL and Related Specification Languages. Ph.D. thesis, Dept. of Computer Science, Univ. of Edinburgh, to appear (1992).

[FJ 90] J. Fitzgerald and C. Jones. Modularizing the formal description of a database system. *Proc. VDM'90 Conference*, Kiel. Springer LNCS 428 (1990).

[Gau 84] M.-C. Gaudel. A first introduction to PLUSS. Technical report, LRI, Université de Paris-Sud, Orsay (1984).

[Gog 84] J. Goguen. Parameterized programming. *IEEE Trans. on Software Engineering* SE-10(5), 528–543 (1984).

[GB 80] J. Goguen and R. Burstall. CAT, a system for the structured elaboration of correct programs from structured specifications. Technical report CSL-118, SRI International (1980).

[GB 84] J. Goguen and R. Burstall. Introducing institutions. *Proc. Logics of Programming Workshop*, Carnegie-Mellon. Springer LNCS 164, 221–256 (1984).

[GB 90] J. Goguen and R. Burstall. Institutions: abstract model theory for specification and programming. Report ECS-LFCS-90-106, Univ. of Edinburgh (1990). *J. ACM*, to appear.

[GM 82] J. Goguen and J. Meseguer. Universal realization, persistent interconnection and implementation of abstract modules. *Proc. Intl. Colloq. on Automata, Languages and Programming*, Aarhus. Springer LNCS 140, 265–281 (1982).

[GM 85] J. Goguen and J. Meseguer. Completeness of many-sorted equational logic. *Houston Journal of Mathematics* 11(3), 307–334 (1985).

[GH 80] J. Guttag and J. Horning. Formal specification as a design tool. *Proc. ACM Symp. on Principles of Programming Languages*, Las Vegas, 251–261 (1980).

[HST 89] R. Harper, D. Sannella and A. Tarlecki. Structure and representation in LF. *Proc. 4th IEEE Symp. on Logic in Computer Science*, Asilomar, 226–237 (1989).

[Hus 85] H. Hußmann. Rapid prototyping for algebraic specifications: RAP system user's manual. Report MIP-8504, Universität Passau (1985).

[Jon 86] C. Jones. *Systematic Software Development using VDM.* Prentice Hall (1986).

[MTH 90] R. Milner, M. Tofte and R. Harper. *The Definition of Standard ML.* MIT Press (1990).

[Mor 90] J. Morris. A methodology for designing and refining specifications. *Proc. 3rd Refinement Workshop*, Hursley Park. Springer BCS Workshop series (1990).

[NO 88] P. Nivela and F. Orejas. Initial behaviour semantics for algebraic specifications. *Selected Papers from the 5th Workshop on Specification of Abstract Data Types*, Gullane. Springer LNCS 332, 184–207 (1988).

[Ore 83] F. Orejas. Characterizing composability of abstract implementations. *Proc. 1983 Intl. Conf. on Foundations of Computation Theory*, Borgholm. Springer LNCS 158 (1983).

[Par 91] H. Partsch. *Requirements Engineering.* Handbuch der Informatik. Oldenbourg (1991).

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

[SB 83] D. Sannella and R. Burstall. Structured theories in LCF. *Proc. Colloq. on Trees in Algebra and Programming*, L'Aquila. Springer LNCS 159, 377–391 (1983).

[SST 90] D. Sannella, S. Sokołowski and A. Tarlecki. Toward formal development of programs from algebraic specifications: parameterisation revisited. Report 6/90, FB Informatik, Universität Bremen (1990). *Acta Informatica*, to appear.

[ST 87] D. Sannella and A. Tarlecki. On observational equivalence and algebraic specification. *J. of Computer and System Sciences* 34, 150–178 (1987).

[ST 88a] D. Sannella and A. Tarlecki. Specifications in an arbitrary institution. *Information and Computation* 76, 165–210 (1988).

[ST 88b] D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: implementations revisited. *Acta Informatica* 25, 233–281 (1988).

[ST 89] D. Sannella and A. Tarlecki. Toward formal development of ML programs: foundations and methodology. *Proc. 3rd Joint Conf. on Theory and Practice of Software Development*, Barcelona. Springer LNCS 352, 375–389 (1989).

[ST 91a] D. Sannella and A. Tarlecki. A kernel specification formalism with higher-order parameterisation. *Proc. 7th Intl. Workshop on Specification of Abstract Data Types*, Wusterhausen. Springer LNCS 534, 274–296 (1991).

[ST 91b] D. Sannella and A. Tarlecki. Extended ML: past, present and future. *Proc. 7th Intl. Workshop on Specification of Abstract Data Types*, Wusterhausen. Springer LNCS 534, 297–322 (1991).

[ST 93] D. Sannella and A. Tarlecki. *Foundations of Algebraic Specifications and Formal Program Development.* Cambridge Univ. Press, to appear (1993?).

[SW 82] D. Sannella and M. Wirsing. Implementation of parameterised specifications. *Proc. Intl. Colloq. on Automata, Languages and Programming*, Aarhus. Springer LNCS 140, 473–488 (1982).

[SW 83] D. Sannella and M. Wirsing. A kernel language for algebraic specification and implementation. *Proc. 1983 Intl. Conf. on Foundations of Computation Theory*, Borgholm. Springer LNCS 158, 413–427 (1983).

[Sch 87] O. Schoett. Data Abstraction and the Correctness of Modular Programming. Ph.D. thesis, report CST-42-87, Dept. of Computer Science, Univ. of Edinburgh (1987).

[Sch 91] O. Schoett. An observational subset of first-order logic cannot specify the behaviour of a counter. *Proc. 8th Symp. on Theoretical Aspects of Computer Science*, Hamburg. Springer LNCS 480, 499–510 (1991).

[Wir 86] M. Wirsing. Structured algebraic specifications: a kernel language. *Theoretical Computer Science* 42, 123–249 (1986).

[Wir 92] M. Wirsing. Structured specifications: syntax, semantics and proof calculus. Technical report, Universität Passau (1992).