Some thoughts on algebraic specification[1,2]

Donald Sannella and Andrzej Tarlecki[3]

Department of Computer Science
University of Edinburgh

1 Introduction

This paper presents in an informal way the main ideas underlying our work on algebraic specification. The central idea, due to Goguen and Burstall, is that much work on algebraic specification can be done independently of the particular logical system (or institution) on which the specification formalism is based. We also examine the nature of specifications and specification languages, the problem of proving that a statement follows from a specification, the important notion of behavioural equivalence, and the evolution of programs from specifications by stepwise refinement. Although many of the issues discussed are motivated by technically complicated problems, in this paper the technicalities have been suppressed in an attempt to make the ideas more accessible. The same ideas are presented with full technical details in [ST 85c].

We assume that the reader is convinced as we are that formal specifications are not only theoretically interesting but are also practically important. Throughout the paper we also assume some familiarity with the basic concepts of algebraic specification, although we do not rely on any specific technical knowledge.

Many of the ideas expressed here were evolved under the influence of Rod Burstall and Martin Wirsing, but this remains a personal statement.

2 Generality and institutions

Any approach to algebraic specification must be based on some logical framework. The pioneering papers [ADJ 76], [Gut 75], [Zil 74] used many-sorted equational logic for this purpose. Nowadays, however, examples of logical systems in use include first-order logic (with and without equality), Horn-clause logic, higher-order logic, infinitary logic, temporal logic and many others. Note that all these logical systems may be considered with or without predicates, admitting partial operations or not. This leads to different concepts of signature and of model, perhaps even more striking in examples like polymorphic signatures, order-sorted signatures, continuous algebras or error algebras.

[1]This research has been supported by a grant from the (U.K.) Science and Engineering Research Council.

[2]Reprinted from: *Proc. 3rd Workshop on Theory and Applications of Abstract Data Types*, Bremen, November 1984, Springer Informatik-Fachberichte Vol. 116, pp. 31-38 (invited note).

[3]Present address: Institute of Computer Science, Polish Academy of Sciences, Warsaw.

There is no reason to view any of these logical systems as superior to the others; the choice must depend on the particular area of application and may also depend on personal taste.

The informal notion of logical system has been formalised by Goguen and Burstall [GB 84], who introduced for this purpose the notion of *institution* (this generalizes the ideas of "abstract model theory" [Bar 74]). An institution defines a notion of a signature together with for any signature Σ a set of Σ-sentences, a collection of Σ-models and a satisfaction relation between Σ-models and Σ-sentences. The only semantic requirement is that when we change signatures, the induced translations of sentences and models preserve the satisfaction relation. This condition expresses the intentional independence of the meaning of specifications from the actual notation. All the above logical systems (and many others) fit into this mould. For example, many-sorted equational logic constitutes an institution: take signatures to be many-sorted algebraic signatures and for any algebraic signature Σ let the set of Σ-sentences be the set of all equations between Σ-terms of the same sort, let Σ-models be just Σ-algebras and let satisfaction of a Σ-equation by a Σ-algebra be defined as usual. It is necessary to adopt some notion of signature morphism $\sigma\colon \Sigma \to \Sigma'$ such as defined e.g. in [BG 80] in order to provide for changing signatures, which induces the σ-translation of Σ-equations to Σ'-equations and of Σ'-algebras to Σ-algebras (σ-reduct).

Note that we can define institutions which diverge from logical tradition and have, for example, sentences expressing constraints on models which are not usually considered in logic, e.g. data constraints as in Clear [BG 80], which impose the requirement of initiality (cf. [Rei 80], [EWT 83]).

For purposes of generality, it is best to avoid choosing any particular logical system on which to base a specification approach, as suggested by Goguen and Burstall. We can instead parameterise our work (whatever it may be) by an *arbitrary* institution. We strongly believe that this is an appropriate level of generality on which to introduce and analyse concepts like specification, implementation, specification-building operations etc. It is possible to define specification languages which can be used to build specifications in any institution (examples are Clear [BG 80] and ASL [ST 85c]).

Of course, not everything can be done in an institution-independent way. For example, a theorem prover needs to know about the detailed structure of axioms. Fixing an institution or even fixing some part of an institution (say, the notion of signature and of model) opens possibilities for doing things which cannot be done at the completely general level. For example, part of a theorem prover (the part dealing with terms, substitutions, type checking etc.) could be built under the single additional assumption that signatures are standard many-sorted algebraic signatures. Striving to always work at the most general level possible results in reusable theories and tools. Typically (but not always) these can be adapted for use under a particular institution simply by providing some low-level details. For example, instantiating the underlying institution in the formal definition of

Clear (and changing the low-level syntax accordingly) yields a family of Clear-like specification languages: equational Clear, error Clear, continuous Clear and so on.

3 Specifications and specification languages

What is a specification? Different views are possible, but one thing which is certain is that a specification is a description of a signature and a class of models over that signature (called the *models of the specification*).

We will not put any restrictions on the class of models described by a specification. Thus, specifications may be *loose* (having non-isomorphic models), so as to avoid premature design decisions. We do not even assume that the class of models of a specification is closed under isomorphism (see [ST 85c] for a brief discussion of this point). In contrast to many approaches (e.g. CIP-L [Bau 81]) we do not require models to be *reachable* (in the standard framework, an algebra is reachable if every element is the value of some ground term; for the generalisation to an arbitrary institution see [Tar 85]). On the other hand, these restrictions are not ruled out, and specification approaches may well contain some mechanism to allow such restrictions to be included in specifications when required (cf. [EWT 83], [ST 85c]).

There are other levels than the level of models at which specifications may be dealt with. For example, we can consider:

Textual level: a sequence of characters on paper,

Presentation level: a signature and a set of axioms over this signature (required to be finite or at least recursive or recursively enumerable),

Theory level: a signature and a set of axioms over this signature closed under logical consequence,

Model level: a signature and a class of models over this signature.

Each approach to specification needs the textual level for actually writing down specifications. The meaning of a specification text is determined by giving a mapping from the textual level to one of the other levels. For example, Clear maps to the theory level, ASL maps to the model level and ACT ONE [EFH 83] to both the presentation and the model level. There are natural mappings from presentations to theories and from theories to classes of models (a presentation maps to the smallest theory containing it, and a theory maps to the class of models satisfying its axioms); the second-level semantics of ACT ONE is actually redundant since it is just the composition of the first-level semantics with these natural mappings, as proved in [EFH 83]. However, not every class of models is the class of models of a theory, and not every theory has a (finite, recursive or recursively enumerable) presentation. In fact, Clear has no presentation-level semantics and ASL has neither a presentation- nor theory-level semantics.

But every specification language has a model-level semantics — and this is in the end all that really matters since the purpose of a specification is not to describe a presentation or a theory but rather to describe a class of models.[4]

Slogan: A specification comprises (at least) a signature and a class of models over this signature.

Everybody knows that big, monolithic specifications are difficult to understand and use. Thus it is important to build specifications in a structured way from small bits. We build specifications in this way using *specification-building operations* (examples: +, **derive**). The semantics of each of these operations is a function on classes of models, e.g. + in ASL corresponds to a function which when given a class of $\Sigma 1$-models and a class of $\Sigma 2$-models yields a class of $(\Sigma 1 \cup \Sigma 2)$-models [SW 83].

A specification language may be viewed as a set of such operations, together with some syntax. Some operations correspond to functions at the presentation or theory level, but in general this need not be so — in any case they are described by functions at the model level.

In choosing the class of operations there is a trade-off between the expressive power of the language and the ease of understanding and dealing with the operations. One way to circumvent this problem is to first develop a *kernel* language which consists of a minimal set of very powerful operations. Such a kernel language is difficult to use directly. We can build higher-level languages on top of the kernel, so that each higher-level construct corresponds to a kernel-language expression. This is analogous to the way that high-level programming languages are defined in terms of machine-level operations. This approach has been taken in ASL; high-level languages built on top of ASL include PLUSS [Gau 84] and Extended ML [ST 85a].

Besides providing a certain collection of predefined specification-building operations, a specification language usually provides a way for the user to define his own specification-building operations, i.e. a mechanism for constructing *parameterised specifications*. There are different approaches to parameterised specifications; the ones which seem most natural in our general framework are those which treat a parameterised specification as a function from specifications to specifications as in e.g. Clear, LOOK [ETLZ 82] or ASL. Such a parameterised specification normally has a certain domain of specifications to which it can be applied.

4 Proving things

In the framework of an arbitrary institution, any class of models determines a theory, that is the set of all sentences which are true in every model belonging

[4]Actually, the ultimate purpose of a specification is typically to describe a class of *programs*, but the notion of a model is chosen so as to precisely capture those aspects of programs which are relevant to the specification while abstracting away from those which are not. For example, in the standard framework algebras are chosen as models in order to abstract away from the syntactic and algorithmic details of programs while capturing their functional behaviour.

to this class (note however that the class of models satisfying this theory may be bigger than the class of models we started with). So every specification determines the set of its logical consequences, the set of sentences which hold in all its models. These are exactly the properties of the specified object expressible in the given institution on which a user is allowed to rely.

In the above, we said nothing about how to effectively determine if a property (sentence) follows from a specification. Our basic notion is the satisfaction relation and model-theoretic (rather than proof-theoretic) consequence. All the same, it would be convenient to have some effective (=computational) way of proving that a sentence is a consequence of a specification, i.e. a *proof system*. This would provide an important tool for the practical use of formal specifications. As suggested by Guttag and Horning [GH 80], by proving that selected properties follow from a specification we can understand it and gain confidence that it expresses what we want. Moreover, in order to do any kind of formal program development or verification a theorem-proving capability is necessary.

Notation: $SP \models \varphi$ means that the sentence φ holds in all models of SP (φ is a semantic consequence of SP). $SP \vdash \varphi$ means that φ is provable from SP in a given proof system.

Any useful proof system must be *sound*, that is $SP \vdash \varphi$ must imply $SP \models \varphi$ (we must only be able to prove things which are true). Another pleasant property is *completeness*, i.e. $SP \models \varphi$ implies $SP \vdash \varphi$ (we can prove all true things). Unfortunately, for every practical specification approach no sound and complete effective proof system can exist; more precisely, this holds for every specification approach which is powerful enough to specify the natural numbers and in which equations can be expressed — see [MS 85] for a discussion of this problem. So we have to be content with a proof system which is sound but not complete. The same situation occurs in program verification; there is no (Cook-) complete Hoare-like proof system for any programming language with a sufficiently rich control structure [Cla 79]. This is too bad, but that's life.

Of course, we cannot expect to be able to construct an institution-independent proof system. We have to assume that we are given some (sound) proof system for the underlying institution, that is a proof system which allows us to deduce sentences from sets of sentences in the institution. This amounts to a proof system for any specification language where specification-building operations are defined at the level of presentations. However, this does not imply that such a semantics is required for doing theorem proving. It is possible to extend the proof system for the underlying institution to a proof system for the specification language in an institution-independent way. What we have to do is to devise an inference rule for every specification-building operation which allows facts about a compound specification to be deduced from facts about its components [SB 83], [ST 85b], [ST 85c]. A simple example of such a rule is:

$$SP \vdash \varphi \implies SP + SP' \vdash \varphi$$

This is another case where, due to the quest for generality via institutions, something (part of a theorem prover for a specification language) may be built once and for all.

5 Behavioural equivalence

A concept which has (not) been extensively (enough) studied in the context of algebraic specifications is that of the behaviour of a program or model. Intuitively, the behaviour of a program is determined just by the answers which are obtained from computations the program may perform. Switching for awhile to the usual algebraic framework, we may say informally that two Σ-algebras are *behaviourally equivalent* with respect to a set *OBS* of *observable sorts* if it is not possible to distinguish between them by evaluating Σ-terms which produce a result of observable sort. For example, suppose Σ contains the sorts *nat*, *bool* and *bunch* and the operations $\mathit{empty}\colon \to \mathit{bunch}$, $\mathit{add}\colon \mathit{nat}, \mathit{bunch} \to \mathit{bunch}$ and $\in\colon \mathit{nat}, \mathit{bunch} \to \mathit{bool}$ (as well as the usual operations on *nat* and *bool*), and suppose A and B are Σ-algebras with

$$\begin{array}{rcl} |A|_{bunch} & = & \text{the set of finite sets of natural numbers} \\ |B|_{bunch} & = & \text{the set of finite lists of natural numbers} \end{array}$$

with the operations and the remaining carriers defined in the obvious way (but B does *not* contain operations like *cons*, *car* and *cdr*). Then A and B are behaviourally equivalent with respect to $\{\mathit{bool}\}$ since every term of sort *bool* has the same value in both algebras (the interesting terms are those of the form $m \in \mathit{add}(a_1, \ldots, \mathit{add}(a_n, \mathit{empty}) \ldots))$. Note that A and B are not isomorphic.

Behavioural equivalence seems to be a concept which is fundamental to programming methodology. For example:

Data abstraction

A practical advantage of using abstract data types in the construction of programs is that the implementation of abstractions by program modules need not be fixed. A different module using different algorithms and/or different data structures may be substituted without changing the rest of the program provided that the new module is behaviourally equivalent to the module it replaces (with respect to the non-encapsulated types). ADJ [ADJ 76] have suggested that “abstract” in “abstract data type” means “up to isomorphism”; we suggest that it really means “up to behavioural equivalence”.

Program specification

One way of specifying a program is to describe the desired input/output behaviour in some concrete way, e.g. by constructing a very simple program which exhibits the desired behaviour. Any program which is behaviourally equivalent to the sample program with respect to the primitive types of the programming language satisfies the specification. This is called an *abstract model specification* [LB 77] or *specification by example* [Sad 84]. In general, specifications under the usual algebraic approaches are not abstract enough; it is either difficult, as in Clear [BG 80] or impossible, as in the initial algebra approach of [ADJ 76] and the final algebra approach of [Wand 79] to specify sets of natural numbers in such a way that both A and B above are models of the specification. ASL provides a *behavioural abstraction* operation which when applied to a specification SP relaxes interpretation to all those algebras which are behaviourally equivalent to a model of SP. We want to stress that although the phrase "specification by example" suggests sloppiness, this is not the case; in this approach it is a precisely-defined, convenient and intuitive way to write specifications, and it is also an established technique in software engineering.

In the above we assume that the only observations (or experiments) we are allowed to perform are to test whether the results of computations are equal. In the context of an arbitrary institution we can generalise this and abstract away from the equational bias by allowing observations which are arbitrary sentences (logical formulae). This yields an institution-independent notion of *observational equivalence*. Two models are observationally equivalent if they both give the same answers to any observation from a prespecified set. Based on this general notion of observational equivalence we can define an institution-independent specification-building operation for observational abstraction (the behavioural abstraction operation mentioned above is actually only a special case of observational abstraction in the standard algebraic framework). The properties of this operation are more complicated than for other specification-building operations, but it is possible to overcome these difficulties and for example to provide proof rules for reasoning about specifications built using observational abstraction [ST 85b].

6 Implementation of specifications

The programming discipline of stepwise refinement suggests that a program be evolved by working gradually via a series of successively lower-level refinements of the specification toward a specification which is so low-level that it can be regarded as a program. For example, the specification

```
reverse(nil) = nil
reverse(cons(a,l)) = append(reverse(l),cons(a,nil))
```

is an executable program in Standard ML [Mil 84]. The stepwise refinement approach guarantees the correctness of the resulting program, provided that each refinement step can be proved correct. A formalisation of this approach requires a precise definition of the concept of refinement, i.e. of the *implementation* of one specification by another.

In programming practice, proceeding from a specification to a program (by stepwise refinement or by any other method) means making a series of design decisions. These will include decisions concerning the concrete representation of abstractly defined data types, decisions about how to compute abstractly specified functions (choice of algorithm) and decisions which select between the various possibilities which the specification leaves open. The following very simple formal notion of implementation captures this idea: a specification SP is implemented by another specification SP', written $SP \rightsquigarrow SP'$, if SP' incorporates more design decisions than SP, i.e. any model of SP' is a model of SP (SP and SP' are required to have the same signature). We can adopt this simple notion in the context of a specification language incorporating an operation like behavioural abstraction (see [SW 83] for more discussion on this point).

This notion of implementation can be extended to give a notion of the implementation of parameterised specifications: P is implemented by P', written $P \rightsquigarrow P'$, if for all specifications SP in the domain of P, SP is also in the domain of P' and $P(SP) \rightsquigarrow P'(SP)$.

An important issue for any notion of implementation is whether implementations can be composed *vertically* and *horizontally* [GB 80]. Implementations can be vertically composed if the implementation relation is transitive ($SP \rightsquigarrow SP'$ and $SP' \rightsquigarrow SP''$ implies $SP \rightsquigarrow SP''$) and they can be horizontally composed if the specification-building operations preserve implementations (i.e. $P \rightsquigarrow P'$ and $SP \rightsquigarrow SP'$ implies $P(SP) \rightsquigarrow P'(SP')$). The above notion of implementation has both these properties, provided that all specification-building operations are monotonic (with respect to inclusion of model classes) which is the case for the specification-building operations defined in e.g. Clear, LOOK and ASL. These two properties allow large structured specifications to be refined in a gradual and modular fashion. All of the individual small specifications which make up a large specification can be separately refined in several stages to give a collection of lower-level specifications (this should be easy because of their small size). When the low-level specifications are put back together, the result is guaranteed to be an implementation of the original specification. Note that other more complicated notions of implementation ([EKMP 82], just to take one example) do not compose vertically or horizontally in general.

7 Final remarks

In this note we put forward some of our thoughts and prejudices concerning algebraic specification. The theme which underlies most of our arguments is one of generality. We argued in favour of working at a high level of generality whenever possible, and against making unnecessary restrictions. The advantage of generality is (at least) the development of reuseable theories and tools; the problem with introducing even the most reasonable-seeming restriction is that we exclude something which we may need someday. An instance of this is the restriction to reachable models (the *generation principle* of [BW 82]) — see [SW 83] for an example which requires considering unreachable models. The result of (for example) fixing an institution at an early stage is that after years of work it is necessary to start again from scratch to introduce some enhancement like the ability to handle higher-order functions or imperative programs.

Practitioners may think we are dreamers because our interest in mathematical elegance and generality seems so far removed from the real world of programmers writing operating systems and payroll programs. But we believe that if practical formal program specification and development is ever to become a reality (and we are optimistic about this) it must be based on sound mathematical foundations. Foundations acceptable in the long term cannot contain restrictions adopted for short-term convenience, e.g. it is a mistake to forever limit yourself to equational specifications because you happen to have a Knuth-Bendix equational theorem prover running on your system. The eventual practical feasibility of all this depends on the existence of good tools for supporting formal program development; although we do not believe that program production will ever be automated, the right tools could reduce the burden of formal program development to an acceptable level.

8 References

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

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

[**Bau 81**] Bauer, F.L. *et al* (the CIP Language Group) Report on a wide spectrum language for program specification and development. Report TUM-I8104, Technische Univ. München; see also: *The wide spectrum language CIP-L.* Springer LNCS 183 (1985).

[**BW 82**] Bauer, F.L. and Wössner, H. *Algorithmic language and program development.* Springer.

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

[Cla 79] Clarke, E.M. Programming language constructs for which it is impossible to obtain good Hoare axiom systems. *J. ACM* 26, 1 pp. 129-147.

[EFH 83] Ehrig, H., Fey, W. and Hansen, H. ACT ONE: an algebraic specification language with two levels of semantics. Report Nr. 83-03, Institut für Software und Theoretische Informatik, Technische Univ. Berlin; see also: Ehrig, H. and Mahr, B. *Fundamentals of Algebraic Specification I: Equations and Initial Semantics*. EATCS Monographs on Theoretical Computer Science, Springer (1985), chapters 9-10.

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

[ETLZ 82] Ehrig, H., Thatcher, J.W., Lucas, P. and Zilles, S.N. Denotational and initial algebra semantics of the algebraic specification language LOOK. Draft report, IBM research.

[EWT 83] Ehrig, H., Wagner, E.G. and Thatcher, J.W. Algebraic specifications with generating constraints. *Proc. 10th Intl. Colloq. on Automata, Languages and Programming*, Barcelona. Springer LNCS 154, pp. 188-202.

[Gau 84] Gaudel, M.-C. A first introduction to PLUSS. Draft report, Univ. de Paris-Sud, Orsay.

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

[GB 84] Goguen, J.A. and Burstall, R.M. Introducing institutions. *Proc. Logics of Programming Workshop*, Carnegie-Mellon. Springer LNCS 164, pp. 221-256.

[Gut 75] Guttag, J.V. The specification and application to programming of abstract data types. Ph.D. thesis, Univ. of Toronto.

[GH 80] Guttag, J.V. and Horning, J.J. Formal specification as a design tool. *Proc. 7th ACM Symp. on Principles of Programming Languages*, Las Vegas, pp. 251-261.

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

[MS 85] MacQueen, D.B. and Sannella, D.T. Completeness of proof systems for equational specifications. *IEEE Transactions on Software Engineering* SE-11, pp. 454-461.

[Mil 84] Milner, R.G. A proposal for Standard ML. *Proc. 1984 ACM Symp. on LISP and Functional Programming*, Austin, Texas.

[Rei 80] Reichel, H. Initially restricting algebraic theories. *Proc. 9th Conf. on Mathematical Foundations of Computer Science*, Rydzyna. Springer LNCS 88, pp. 504-514.

[Sad 84] Sadler, M. Mapping out specification. Draft report, Dept. of Computing, Imperial College, London; presented at: Workshop on Formal Aspects of Specification, Swindon.

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

[ST 85a] Sannella, D.T. and Tarlecki, A. Program specification and development in Standard ML. *Proc. 12th ACM Symp. on Principles of Programming Languages*, New Orleans, pp. 67-77.

[ST 85b] Sannella, D.T. and Tarlecki, A. On observational equivalence and algebraic specification. To appear in *J. Computer and System Sciences*; extended abstract in: *Proc. 10th Colloq. on Trees in Algebra and Programming*, Joint Conf. on Theory and Practice of Software Development (TAPSOFT), Berlin. Springer LNCS 185, pp. 308-322.

[ST 85c] Sannella, D.T. and Tarlecki, A. Specifications in an arbitrary institution. To appear in *Information and Control*; see also: Sannella, D.T. and Tarlecki, A. Building specifications in an arbitrary institution. *Proc. Intl. Symposium on Semantics of Data Types*, Sophia-Antipolis. Springer LNCS 173, pp. 337-356.

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

[Tar 85] Tarlecki, A. On the existence of free models in abstract algebraic institutions. *Theoretical Computer Science* 37, pp. 269-304.

[Wand 79] Wand, M. Final algebra semantics and data type extensions. *J. Computer and System Sciences* 19, pp. 27-44.

[Zil 74] Zilles, S.N. Algebraic specification of data types. Computation Structures Group memo 119, Laboratory for Computer Science, MIT.