Algebraic specification and program development by stepwise refinement*

Extended abstract

Donald Sannella

Laboratory for Foundations of Computer Science
University of Edinburgh, UK

`dts@dcs.ed.ac.uk` `www.dcs.ed.ac.uk/~dts/`

Abstract. Various formalizations of the concept of "refinement step" as used in the formal development of programs from algebraic specifications are presented and compared.

1 Introduction

Algebraic specification aims to provide a formal basis to support the systematic development of correct programs from specifications by means of verified refinement steps. Obviously, a central piece of the puzzle is how best to formalize concepts like "specification", "program" and "refinement step". Answers are required that are simple, elegant and general and which enjoy useful properties, while at the same time taking proper account of the needs of practice. Here I will concentrate on the last of these concepts, but first I need to deal with the other two.

For "program", I take the usual approach of algebraic specification whereby programs are modelled as *many-sorted algebras* consisting of a collection of sets of data values together with functions over those sets. This level of abstraction is commensurate with the view that the correctness of the input/output behaviour of a program takes precedence over all its other properties. With each algebra is associated a *signature* Σ which names its components (sorts and operations) and thus provides a basic vocabulary for making assertions about its properties. There are various definitions of signature and algebra but the details will not be important here. The class of Σ-algebras is denoted $Alg(\Sigma)$.

For "specification", it will be enough to know that any specification SP determines a signature $Sig(SP)$ and a class $[\![SP]\!]$ of $Sig(SP)$-algebras. These algebras (the *models* of SP) correspond to all the programs that we regard as correct realizations of SP. Algebraic specification is often referred to as a "property-oriented" approach since specifications contain *axioms*, usually in some flavour of first-order logic with equality, describing the properties that models are required to satisfy. But again, the details of what specifications look like will not

* This research was supported by EPSRC grant GR/K63795 and the ESPRIT-funded CoFI Working Group.

concern us here. Sometimes SP will tightly constrain the behaviour of allowable realizations and $[\![SP]\!]$ will be relatively small, possibly an isomorphism class or even a singleton set; other times it will impose a few requirements but leave the rest unconstrained, and then $[\![SP]\!]$ will be larger. We allow both possibilities; in contrast to approaches to algebraic specification such as [EM85], the "initial model" of SP (if there is one) plays no special rôle.

The rest of this paper will be devoted to various related formalizations of the concept of "refinement step". I use the terms "refinement" and "implementation" interchangeably to refer to a relation between specifications, while "realization" is a relation between an algebra or program and a specification. An idea-oriented presentation of almost all of this material, with examples, can be found in [ST97] and this presentation is based on that. See [ST88], [SST92], [BST99] and the references in [ST97] for a more technical presentation. Someday [ST??] will contain a unified presentation of the whole picture and at that point everybody reading this must immediately go out and buy it. Until then, other starting points for learning about algebraic specification are [Wir90], [LEW96] and [AKK99].

2 Simple refinement

Given a specification SP, the programming task it defines is to construct an algebra (i.e. program) A such that $A \in [\![SP]\!]$. Rather than attempting to achieve this in a single step, we proceed systematically in a stepwise fashion, incorporating more and more design and implementation decisions with each step. These include choosing between the options of behaviour left open by the specification, between the algorithms that realize this behaviour, between data representation schemes, etc. Each such decision is recorded as a separate step, typically consisting of a local modification to the specification. Developing a program from a specification then involves a sequence of such steps:

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

Here, SP_0 is the original specification of requirements 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) that is an exact description of an algebra.

A formal definition of $SP \rightsquigarrow SP'$ must incorporate the requirement that any realization of SP' is a correct realization of SP. This gives [SW83,ST88]:

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

which presupposes that $Sig(SP) = Sig(SP')$. This is the *simple refinement* relation.

Stepwise refinement is sound precisely because the correctness of the final outcome can be inferred from the correctness of the individual refinement steps:

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

In fact, the simple refinement relation is transitive:

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

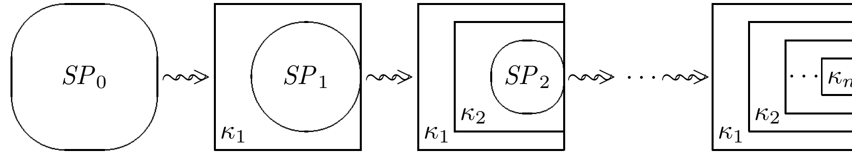
Typically, the specification formalism will contain operations for building complex specifications from simpler ones. If these operations are monotonic w.r.t. inclusion of model classes (this is a natural requirement that is satisfied by almost all specification-building operations that have ever been proposed) then they preserve simple refinement:

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

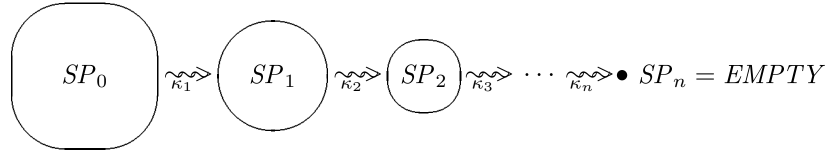
This provides one way of decomposing the task of realizing a structured specification into a number of separate subtasks, but it unrealistically requires the structure of the final realization to match the structure of the specification. See Sect. 4 below for a better way.

3 Constructor implementation

In the context of a sufficiently rich specification language, simple refinement is powerful enough to handle all concrete examples of interest. However, it is not very convenient to use in practice. During stepwise refinement, the successive specifications accumulate more and more details arising from successive design decisions. 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 separate the finished parts from the specification, proceeding with the development of the unresolved parts only.



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 act as constructions

extending any realization of the unresolved part to a realization of what is being refined.

Each κ_i amounts to a so-called *parameterised program* [Gog84] with input interface SP_i and output interface SP_{i-1}, or equivalently a *functor* in Standard ML. I call it a *constructor*, not to be confused with value constructors in functional languages. Semantically, it is a function on algebras $\kappa_i : Alg(Sig(SP_i)) \rightarrow Alg(Sig(SP_{i-1}))$. Intuitively, κ_i provides a definition of the components of a $Sig(SP_{i-1})$-algebra, given the components of a $Sig(SP_i)$-algebra.

Constructor implementation [ST88] is defined as follows. Suppose that SP and SP' are specifications and κ is a constructor such that $\kappa : Alg(Sig(SP')) \rightarrow Alg(Sig(SP))$. Then:

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

(Here, $\kappa([\![SP']\!])$ is the image of $[\![SP']\!]$ under κ.) We read $SP \underset{\kappa}{\rightsquigarrow} SP$ as "SP' implements SP via κ".

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

$$\frac{SP_0 \underset{\kappa_1}{\rightsquigarrow} SP_1 \underset{\kappa_2}{\rightsquigarrow} \cdots \underset{\kappa_n}{\rightsquigarrow} SP_n = EMPTY}{\kappa_1(\kappa_2(\ldots \kappa_n(empty)\ldots)) \in [\![SP_0]\!]}$$

where $EMPTY$ is the empty specification over the empty signature and *empty* is its (empty) realization. Again, the constructor implementation relation is in fact transitive:

$$\frac{SP \underset{\kappa}{\rightsquigarrow} SP' \qquad SP' \underset{\kappa'}{\rightsquigarrow} SP''}{SP \underset{\kappa \circ \kappa'}{\rightsquigarrow} SP''}$$

4 Problem decomposition

Decomposition of a programming task into separate subtasks is modelled using a constructor implementation with a multi-argument constructor [SST92]:

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

where $\kappa : Alg(Sig(SP_1)) \times \cdots \times Alg(Sig(SP_n)) \rightarrow Alg(Sig(SP))$ is an n-argument constructor. Now the development takes on a tree-like shape. It is complete once a tree is obtained that has empty sequences (of specifications) as its leaves:

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

Then an appropriate instantiation of the constructors in the tree yields a realization of the original requirements specification. The above development tree yields the algebra

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

The structure of the final realization is determined by the shape of the development tree, which is in turn determined by the decomposition steps. This is in contrast to the naive form of problem decomposition mentioned earlier, where the structure of the final realization is required to match the structure of the specification.

5 Behavioural implementation

A specification should not include unnecessary constraints, even if they happen to be satisfied by a possible future realization, since this may prevent the developer from choosing a different implementation strategy. 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 received considerable attention, see e.g. [BHW95]. 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. 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, i.e. terms of sorts in OBS with variables (representing the inputs) of sorts in OBS only. Two Σ-algebras A and B are *behaviourally equivalent* (w.r.t. OBS), written $A \equiv B$, if all observable computations yield the same results in A and in B.

It turns out to be difficult to write specifications having model classes that are closed under behavioural equivalence, largely because of the use of equality in axioms. One solution is to define $[\![\cdot]\!]$ such that $[\![SP]\!]$ always has this property, but this leads to difficulties in reasoning about specifications. Another is to take account of behavioural equivalence in the notion of implementation.

Behavioural implementation [ST88] is defined as follows. Suppose that SP and SP' are specifications and κ is a constructor such that $\kappa : Alg(Sig(SP')) \to Alg(Sig(SP))$. Then:

$$SP \overset{\equiv}{\underset{\kappa}{\leadsto}} SP' \quad \textbf{iff} \quad \forall A \in [\![SP']\!]. \exists B \in [\![SP]\!]. \kappa(A) \equiv B$$

This is just like constructor implementation except that κ applied to a model of SP' is only required to be a model of SP modulo behavioural equivalence.

A problem with this definition is that stepwise refinement is unsound. The following property does *not* hold:

$$\frac{SP_0 \overset{\equiv}{\underset{\kappa_1}{\leadsto}} SP_1 \overset{\equiv}{\underset{\kappa_2}{\leadsto}} \cdots \overset{\equiv}{\underset{\kappa_n}{\leadsto}} SP_n = EMPTY}{\exists A \in [\![SP_0]\!]. \kappa_1(\kappa_2(\ldots \kappa_n(empty) \ldots)) \equiv A}$$

The problem is that $SP_0 \underset{\kappa_1}{\overset{\equiv}{\rightsquigarrow}} SP_1$ ensures only that algebras in $[\![SP_1]\!]$ give rise to correct realizations of SP_0. It says nothing about the algebras that are only models of SP_1 up to behavioural equivalence. But such algebras may arise as well because $SP_1 \underset{\kappa_2}{\overset{\equiv}{\rightsquigarrow}} SP_2$.

The problem disappears if we modify the definition of behavioural implementation $SP \underset{\kappa}{\overset{\equiv}{\rightsquigarrow}} SP'$ to require

$$\forall A \in Alg(Sig(SP')).(\exists A' \in [\![SP']\!].A \equiv A') \Rightarrow (\exists B \in [\![SP]\!].\kappa(A) \equiv B)$$

but then it is very difficult to prove the correctness of behavioural implementations. There is a better way out, originally suggested in [Sch87]. Soundness of stepwise refinement using our original definition of behavioural implementation is recovered, as well as transitivity of the behavioural implementation relation, if we assume that all constructors used are *stable*, that is, that any constructor $\kappa : Alg(Sig(SP')) \rightarrow Alg(Sig(SP))$ preserves behavioural equivalence:

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

We could repeat here the tree-like development picture of Sect. 4 — developments involving decomposition steps based on behavioural implementations with multi-argument (stable) constructors yield correct realizations of the original requirements specification.

There are two reasons why stability is a reasonable assumption. First, recall that constructors correspond to parameterised programs which means that they must be written in some given programming language. The stability of *expressible* constructors can be established in advance for this programming language, and this frees the programmer from the need to prove it during the program development process. Second, 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 [Sch87] we view the stability requirement as a methodologically justified design criterion for the modularisation facilities of programming languages.

6 Refinement steps in Extended ML and CASL

The presentation above may be too abstract to see how the ideas apply to the development of concrete programs. It may help to see them in the context of a particular specification and/or programming language.

Extended ML [San91,KST97] is a framework for the formal development of Standard ML programs from specifications. Extended ML specifications look just like Standard ML programs except that axioms are allowed in "signatures" (module interface specifications) and in place of code in module bodies. As noted above, constructors correspond to Standard ML functors. Extended ML functors,

with specifications in place of mere signatures as their input and output interfaces, correspond to constructor implementation steps: the well-formedness of `functor F(X:`SP`):`SP'` = body` in Extended ML corresponds to the correctness of $SP' \underset{\mathtt{F}}{\rightsquigarrow\!\!>} SP$. There is a close connection with the notion of *steadfast program* in [LOT99]. Extended ML functors are meant to correspond to behavioural implementation steps and the necessary underlying theory for this is in [ST89], but the requisite changes to the semantics of Extended ML are complicated and have not yet been satisfactorily completed. The Extended ML formal development methodology accommodates stepwise refinement with decomposition steps as above, generalized to accommodate development of functors as well as structures (algebras).

CASL, the new *Common Algebraic Specification Language* [CoFI98], has been developed under the auspices of the Common Framework Initiative [Mos97] in an attempt to consolidate past work on the design of algebraic specification languages and provide a focal point for future joint work. *Architectural specifications* in CASL [BST99] relate closely to constructor implementations in the following sense. Consider $SP \underset{\kappa}{\rightsquigarrow\!\!>} \langle SP_1, \ldots, SP_n \rangle$ where κ is a multi-argument constructor. The architectural specification

`arch spec ASP = units` U_1`:`SP_1`; ... ;` U_n`:`SP_n `result` T

(where T is a so-called *unit term* which builds an algebra from the algebras $U_1, \ldots, U_n$) includes $SP_1, \ldots, SP_n$ and $\kappa = \lambda U_1, \ldots, U_n.T$ but not SP. Its semantics is (glossing over many details) the class $\kappa([\![SP_1]\!] \times \cdots \times [\![SP_n]\!])$. Thus $SP \underset{\kappa}{\rightsquigarrow\!\!>} \langle SP_1, \ldots, SP_n \rangle$ corresponds to the simple refinement $SP \rightsquigarrow\!\!> \mathtt{ASP}$. CASL accommodates generic units so it also allows development of parameterised programs.

7 Higher order extensions

Algebraic specification is normally restricted to first-order. There are three orthogonal dimensions along which the picture above can be extended to higher-order.

First, we can generalize constructor implementations by allowing constructors to be higher-order parameterised programs. If we extend the specification language to permit the specification of such programs (see [SST92,Asp97]) then we can develop them stepwise using the definitions of Sect. 3, with decomposition as in Sect. 4. Both Extended ML and CASL support the development of first-order parameterised programs. In both cases the extension to higher-order parameterised programs has been considered but not yet fully elaborated. Higher-order functors are available in some implementations of Standard ML, cf. [Rus98]. To apply behavioural implementation, one would require an appropriate notion of behavioural equivalence between higher-order parameterised programs.

Second, we can use higher-order logic in axioms. Nothing above depends on the choice of the language of axioms, but the details of the treatment of

behavioural equivalence is sensitive to this choice. The treatment in [BHW95] extends smoothly to this case, see [HS96].

Finally, we can allow higher-typed functions in signatures and algebras. Again, the only thing that depends on this is the details of the treatment of behavioural equivalence. Behavioural equivalence of such algebras is characterized by existence of a so-called *pre-logical relation* between them [HS99]. If constructors are defined using lambda calculus then stability is a consequence of the Basic Lemma of pre-logical relations [HLST00].

Acknowledgements: Hardly any of the above is new, and all of it is the product of collaboration. Thanks to Martin Wirsing for starting me off in this direction in [SW83], to Andrzej Tarlecki (especially) for close collaboration on most of the remainder, to Martin Hofmann for collaboration on [HS96], to Furio Honsell for collaboration on [HS99], and to Andrzej, Furio and John Longley for collaboration on [HLST00]. Finally, thanks to the LOPSTR'99 organizers for the excuse to visit Venice.

References

[Asp97] D. Aspinall. Type Systems for Modular Programs and Specifications. Ph.D. thesis, Dept. of Computer Science, Univ. of Edinburgh (1997).

[AKK99] E. Astesiano, H.-J. Kreowski and B. Krieg-Brückner (eds.). *Algebraic Foundations of Systems Specification*. Springer (1999).

[BHW95] M. Bidoit, R. Hennicker and M. Wirsing. Behavioural and abstractor specifications. *Science of Computer Programming* 25:149–186 (1995).

[BST99] M. Bidoit, D. Sannella and A. Tarlecki. Architectural specifications in Casl. *Proc. 7th Intl. Conference on Algebraic Methodology and Software Technology (AMAST'98)*, Manaus. Springer LNCS 1548, 341–357 (1999).

[CoFI98] CoFI Task Group on Language Design. Casl – The CoFI algebraic specification language – Summary (version 1.0). `http://www.brics.dk/Projects/CoFI/Documents/CASL/Summary/` (1998).

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

[Gog84] J. Goguen. Parameterized programming. *IEEE Trans. on Software Engineering* SE-10(5):528–543 (1984).

[HS96] M. Hofmann and D. Sannella. On behavioural abstraction and behavioural satisfaction in higher-order logic. *Theoretical Computer Science* 167:3–45 (1996).

[HLST00] F. Honsell, J. Longley, D. Sannella and A. Tarlecki. Constructive data refinement in typed lambda calculus *Proc. 3rd Intl. Conf. on Foundations of Software Science and Computation Structures*. European Joint Conferences on Theory and Practice of Software (ETAPS 2000), Berlin. Springer LNCS 1784, 149–164 (2000).

[HS99] F. Honsell and D. Sannella. Pre-logical relations. *Proc. Computer Science Logic, CSL'99*, Madrid. Springer LNCS 1683, 546–561 (1999).

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

[LOT99] K.-K. Lau, M. Ornaghi and S.-Å. Tärnlund. Steadfast logic programs. *Journal of Logic Programming* 38:259–294 (1999).

[LEW96] J. Loeckx, H.-D. Ehrich and M. Wolf. *Specification of Abstract Data Types*. Wiley (1996).

[Mos97] P. Mosses. CoFI: The Common Framework Initiative for algebraic specification and development. *Proc. 7th Intl. Joint Conf. on Theory and Practice of Software Development*, Lille. Springer LNCS 1214, 115–137 (1997).

[Rus98] C. Russo. Types for Modules. Ph.D. thesis, report ECS-LFCS-98-389, Dept. of Computer Science, Univ. of Edinburgh (1998).

[San91] D. Sannella. Formal program development in Extended ML for the working programmer. *Proc. 3rd BCS/FACS Workshop on Refinement*, Hursley Park. Springer Workshops in Computing, 99–130 (1991).

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

[ST88] 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. *Proc. 3rd Joint Conf. on Theory and Practice of Software Development*, Barcelona. Springer LNCS 352, 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).

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

[SW83] 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).

[Sch87] 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).

[Wir90] M. Wirsing. Algebraic specification. *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.). North-Holland (1990).