Algebraic methods for specification and formal development of programs[1]

Donald Sannella
Laboratory for Foundations of Computer Science, University of Edinburgh. E-mail: dts@dcs.ed.ac.uk
and
Andrzej Tarlecki
Institute of Informatics, Warsaw University and Institute of Computer Science, Polish Academy of Sciences, Warsaw. E-mail: tarlecki@mimuw.edu.pl.

This note gives our personal perspective on the state of foundations of software specification and development including applications to the formal development of reliable complex software systems. We regard this area of research as straddling the borderline between theory and practice. It has connections with work on the design and semantics of software systems and programming languages, on formal methods for system verification (various program logics in particular), and relies heavily on some basic concepts of mathematical logic, universal algebra and category theory, while being directly inspired by and potentially applicable in practice.

In this note we sum up our experiences so far, hint at the need for further work in certain areas, and speculate a bit about the directions in which we expect to go in the future.

The roots of our work in this area are in the theory of algebraic specification. The most fundamental assumption underlying this theory is that 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. Another common element is that specifications of programs consist mainly of logical *axioms*, usually in a logical system in which equality has a prominent role, describing the properties that the functions are required to satisfy. This *property-oriented* approach is in contrast to so-called *model-oriented* specifications which consist of a simple realization of the required behaviour. A wide variety of different approaches to algebraic specification take these two principles as their starting point.

Research on algebraic specification has been devoted mainly to the search for an adequate account of the fundamental concepts and basic processes involved in the specification and formal development of programs. Some of the key questions are: What is a specification? What does a specification mean? When does a program satisfy a specification? When does a specification guarantee a property that it does not state explicitly? How does one prove this? How are specifications structured? How does the structure of specifications relate to the structure of programs? When does one specification correctly refine another specification? How does one prove correctness of refinement steps? When do refinement steps compose? What is the role of information hiding? Answers are required that are simple, elegant and general while at the same time taking some account of the needs of practice. In our view, most of the elements of a satisfactory foundational account are now in place; see [Sannella and Tarlecki 1997] for an overview with pointers to the literature. An important characteristic of this theory is that its treatment of modular structure is *compositional*, allowing large examples to be treated by decomposition into smaller units.

The original motivation for work on algebraic specification was to provide support for the systematic development of correct programs from specifications with verified development steps. An adequate theoretical basis is a prerequisite to achieving this goal, but it is not enough in itself. The theoretical work is motivated by the problems of correctness of real software, and the foundational concepts are inspired by the constructs found in programming languages, but the concrete link between the foundations and actual programming languages used for developing actual software systems remains rather tenuous.

Our contribution to filling this gap is our work on Extended ML (EML), which we have pursued since 1985 in parallel to our work on foundational issues [Kahrs et al. 1997]. Our aim here was (we thought) relatively modest: to apply the emerging theory of algebraic specification to the formal development of modular Standard ML (SML) programs from specifications of their required behaviour. SML seemed to us to be an ideal choice of programming language for such an enterprise. It is a functional language, which fits comfortably with the philosophical view of work on algebraic specification. It has an advanced module system which seemed (at that time, at least) tailor-made for the support of program development by decomposition into reusable generic modules with well-defined interfaces, with a clear separation between the "in the small" part of the language and the "in the large" part which exactly matches a stratification that can usefully be imposed in the theory of algebraic specifications using so-called *institutions*. SML is a "real" programming language in the sense that it is used for programming applications rather than merely as an object of study, and it is almost unique among such languages in having a formal semantics. Finally, it appears to be small and elegant enough to admit analysis, at least in principle. If you can't do it for SML, we reasoned, it can't be done.

The outcome of our research on EML leaves us with mixed feelings. On the positive side, we discovered strengths and weaknesses in the SML module system for use in our context, some of which were later re-discovered by other people in other contexts, see e.g. [Leroy 1994]. We invented a methodology for the gradual development of modular systems from specifications which seems very smooth. We produced a complete formal semantics for EML [Kahrs et al. 1997], building on the semantics of SML. EML has proved to be useful in teaching, with relatively few new concepts or new notations to learn for people who have already been exposed to SML. In Edinburgh it is used as a vehicle for teaching software design as well as to introduce formal methods. Students can write specifications and formally develop programs of a few pages in length from them without too much difficulty, provided one restricts to a simple subset of SML. And our work on EML has inspired novel foundational research. The most prominent example of this is our work in [Sannella et al. 1992] on the difference between specifications of generic modules and parameterised specifications. Applying ideas on institutions in the EML context allowed us to make a start on applying the ideas of EML in the context of other programming languages, see e.g. [Read and Kazmierczak 1992] for an experiment in formal development of modular Prolog programs. On the negative side, the semantics of EML is more complicated than we had expected. See [Kahrs and Sannella 1998] for some reasons why this seems to be necessarily so. This is reflected in the difficulty of proving properties of programs and specifications. Again, this seems necessarily difficult, see e.g. [Pitts and Stark 1993]. In order to provide extra flexibility, EML (soundly) allows code to satisfy specifications "up to behavioural equivalence" rather than requiring literal satisfaction; this turns out to be harder to treat than we had expected, but on the other hand it seems to arise less frequently in examples than we thought it would. The institutional view of EML works up to a point, but it seems difficult to really push it through all the way [Sannella and Tarlecki 1986]. Actually writing specifications of programs that use higher-order functions, exceptions, partial functions etc. is difficult and the resulting specifications are hard to read. Worse, the specifications of simple total first-order functions need to include boring conditions that rule out exceptions and partiality. Finally, actually developing programs from specifications is hard work.

In retrospect, our goal was too ambitious. SML contains features that have not been satisfactorily treated by work on specification and verification even when taken in isolation, not to mention when taken in combination. Real programming languages are inevitably complex, and any serious attempt to give a formal treatment of such a language and a development framework based on it is an ambitious undertaking bringing a host of problems that do not arise when considering toy programming languages or when considering specification and formal development in abstract terms. Our EML experience suggests that, at least at the present time, tackling the problems of specification and formal development in a real programming language at a fully formal level is just too difficult. This seems to leave two ways forward: either remain fully formal but focus on a smaller and simpler language, perhaps building up from this gradually to something approaching a real language; or remain with a real language, but give up trying to achieve full formality. It seems that the proper arena for dealing with the real problems of practical development of correct software is the latter, while the former is required if we want to obtain a deeper understanding of the link between programming and the theory of formal development. In the less formal strand the aim is to make programming more efficient and to produce better programs rather than to provide guarantees of correctness. If proofs are undertaken at all, they are viewed as a cost-effective way of uncovering errors rather than as a way of ruling out the possibility of error. If guarantees are required, the framework provided by the more formal strand is available whenever the problem at hand can be addressed within the restrictions imposed by the simpler language studied there. These two strands of work are complementary and have a great deal to contribute to each other; ideally, the simple language studied in one might be embedded in the larger language studied in the other, with some of the results, experience and methods obtained in each strand being applicable in the other via the embedding. We would like to think that they might one day come together to give a synthesis that fully achieves the original goals of our work on EML.

By focusing on SML and the problem of achieving correctness with respect to a given specification of requirements, we knew from the beginning that work on EML disregarded many important issues. We reasoned that the problems that we were attacking would need to be solved too, and that we could worry about additional complications once we had solved these. It seems that the time has come to attack some of the outstanding issues now, perhaps following the less formal strand of work as discussed above in order to enable early application. One class of shortcomings stem from restrictions that are inherited from SML. Since SML is a sequential language, we do not address the problem of correctness of concurrent systems, despite the fact that this is where the benefit of formal analysis appears to really pay off. The same goes for other programming paradigms: for example, there is no proper treatment of objects in SML even though many of the elements of object-oriented programming are present. Adding a treatment of concurrency is an attractive avenue for future work, but it is unclear to us how to proceed because we see no approach to building concurrent systems “in the large” that compares with ML’s module system for sequential programs. A second class of shortcomings stem from our focus on a specific part of the program development process, and here the criticism applies to most work on algebraic specification. One example is the crucial problem of where the original specification of requirements comes from and how to ensure that it accurately reflects the needs of the intended user. This phase, which we ignore in EML because it involves unformalizable elements, is regarded by many software engineers as a more critical phase in the pursuit of quality software than the design and coding phases we focus on. In the context of modular systems the problem appears to be larger because of the need to formulate interface specifications of all the modules in the program, not just of the program as a whole. But the difference is that here everything is formal, so there is a precise sense in which interfaces need to be compatible when modules are interconnected, and this can be checked.

Despite our difficulties with EML, the algebraic approach still seems very attractive to us. The existing theory is elegant and has at least some of the attributes (e.g. compositionality) required for its application to practice to succeed. Even if the application of this theory is less direct than we had originally hoped, the clarity of concepts that it provides allows it to serve as a useful abstract reference model for some aspects of practical software development. Comparing emerging practical methods (e.g. methods for decomposing problems and composing solutions, or for checking the compatibility of design decisions) against this yardstick might provide a basis for judging their soundness and/or limitations. The practical problems that inspired the development of the theory of algebraic specification still await satisfactory solutions. Some argue that other issues are more critical; we don’t disagree with this, but the problem of going from a formal specification to a program that satisfies it is an interesting and well-defined one which deserves attention too. And the solution to this problem is fundamental in the sense that one criterion for the quality of solutions to other problems (e.g. strategies for testing, or methods for requirements capture or rapid prototyping) is the extent to which they fit into this formal picture.

REFERENCES

KAHRS, S. AND SANNELLA, D. 1998. Reflections on the design of a specification language. In *Proc. Intl. Colloq. on Fundamental Approaches to Software Engineering. European Joint Conferences on Theory and Practice of Software (ETAPS'98)*, Volume 1382 of *Lecture Notes in Computer Science* (1998), pp. 154–170. Springer.

KAHRS, S., SANNELLA, D., AND TARLECKI, A. 1997. The definition of Extended ML: A gentle introduction. *Theoretical Computer Science 173*, 445–484.

LEROY, X. 1994. Manifest types, modules, and separate compilation. In *Proc. 21st ACM Symp. on Principles of Programming Languages* (1994), pp. 109–122. ACM Press.

PITTS, A. AND STARK, I. 1993. Observable properties of higher order functions that dynamically create local names, or: What's new? In *Proc. 18th Intl. Symp. on Mathematical Foundations of Computer Science*, Volume 711 of *Lecture Notes in Computer Science* (1993), pp. 122–141. Springer.

READ, M. AND KAZMIERCZAK, E. 1992. Formal program development in modular Prolog: A case study. In *Proc. Workshop on Logic Program Synthesis and Transformation*, Workshops in Computing (1992), pp. 69–93. Springer.

SANNELLA, D., SOKOŁOWSKI, S., AND TARLECKI, A. 1992. Toward formal development of programs from algebraic specifications: Parameterisation revisited. *Acta Informatica 29*, 8, 689–736.

SANNELLA, D. AND TARLECKI, A. 1986. Extended ML: An institution-independent framework for formal program development. In *Proc. Workshop on Category Theory and Computer Programming*, Volume 240 of *Lecture Notes in Computer Science* (1986), pp. 364–389. Springer.

SANNELLA, D. AND TARLECKI, A. 1997. Essential concepts of algebraic specification and program development. *Formal Aspects of Computing 9*, 229–269.