Algebraic specification and formal methods for program development: what are the real problems?

Donald Sannella* Andrzej Tarlecki†

1 Introduction

The long-term goal of work on algebraic specification is formal development of correct programs from specifications via verified-correct refinement steps. Define a *real problem* to be an unsolved problem which lies on (or near) the path between the current state of the art and this ultimate goal. Long-term progress depends on solving these problems, so it seems worthwhile to attack the real problems before worrying about other issues.

It is perhaps surprising that there is little agreement concerning what these problems are, at least if one takes the problems being tackled as an indication of what various researchers think the real problems are. Some sort of consensus seems desirable to promote effective joint progress towards our common goal.

We list below some (not all) of what we think are the real problems. In an attempt to spark controversy, some things which we think are *not* real problems are also listed. Neither of these lists is exhaustive.

2 Some real problems and some non-problems

The process of developing a program from a specification begins with a requirements specification. The first issue is how this requirements specification comes into being. This is the topic of requirements engineering, which is as yet in its infancy. Much work is needed here; the central problem is how to bridge the gap between the completely informal ideas and goals of the customer and the completely formal language of algebraic specifications.

Problem Requirements engineering: evolving accurate formal requirements specifications from informal ideas.

Once a version of the requirements specification exists, it is necessary to ensure that it reflects the customer's real intentions and needs. There are two aspects of this problem: the methodological one of knowing what to check and phrasing the property to be checked in an appropriate formalism, and the technical one of checking whether a formal requirements specification ensures such a property. Theorem-proving technology, needed elsewhere in the program development process anyway, solves the latter problem if the property is expressed as a logical formula.

Problem Specification testing: checking that a requirements specification reflects the customer's intentions.

A traditional approach to solving this problem is to force the specification to be written in an "executable" specification language so that the customer can run examples to check if the results obtained

*FB3 — Mathematik/Informatik, Universität Bremen, Bremen, West Germany.
†Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland.

are the desired ones. We think that this is a bad idea. There is a fundamental tradeoff between executability and expressiveness, and it is clearly the latter which is of central importance in a language intended for writing requirements specifications. Demanding that requirements specifications be written in an executable specification language is not much different from requiring that they be written in a high-level programming language like ML. As a result, most of the alleged benefits of formally developing programs from specifications are sacrificed before the development process has even begun.

Non-problem Making requirements specifications executable.

Perhaps it would be more accurate to say that this is a *non-solution* to the problem of specification testing.

All specifications which arise during the development process need to be built in a structured way from relatively small units. Work on building structured specifications has concentrated both on foundational aspects of the problem (semantics, elementary structuring mechanisms, etc.) and on the design of user-friendly specification languages and notations. Languages have been proposed with various combinations of features. The design of such languages and their foundations is a non-problem in the sense that there are a number of well-developed approaches. Nevertheless, we list it as a problem since none of these approaches is manifestly perfect and so other approaches are worth exploring.

Problem Design of a "perfect" specification language.

The structure of a specification incorporates intangible aspects of the specifier's knowledge of the problem being specified and is essential to the understanding of the specification. This structure is not only a matter of presentation but can play an important role in its use. For this reason, dealing with structured specifications by "normalizing out" the structure (if this is an option) is to be avoided when possible. It is not always possible to convert specifications to some simple normal form, for example to a flat specification; this is not a problem since there is no good reason to do this.

Non-problem Normalizing structured specifications to unstructured specifications.

This is related to the issue of normalizing structure on other levels, for example the structure of an implementation or the modular structure of a system. We see no practical reason to care whether or not various ways of composing things give things which can be presented in some standard form. This composition must be meaningful on the semantic level (given the semantics of component presentations, the semantics of the composition of these presentations must be well-defined) but it is not necessary to insist that this composition has a presentation in the same form as the components.

Non-problem Syntactic composability of module presentations, implementation presentations, etc.

Specification languages and software development frameworks should be made independent from the particular logical system used to write axioms insofar as this is possible. This is desirable in order to allow different logics to be used for different purposes, thereby broadening the range of problems to which such languages and frameworks may be applied. It follows that their foundations should be expressed in logic-independent terms, which may be accomplished by working within an arbitrary logical system encoded (for example) as a so-called institution.

Methods of defining specification languages and other aspects of software development frameworks in the context of an arbitrary institution are relatively well-established. What is not so clear is the relationship between model-theoretic formulations of general logic (such as institutions) and proof-theoretic formulations (such as Edinburgh LF or Isabelle). Appropriate connections are required in make full use of theorem-proving tools being developed for the proof-theoretic approaches in program development frameworks based on model-theoretic approaches.

Problem Establishing satisfactory connections between model-theoretic and proof-theoretic formulations of general logic.

The motivation for developing systems which are not dependent on any particular logical system is that no single logical system is clearly adequate for all purposes. In most practical situations it is necessary to use heterogeneous logics which contain constructs for dealing with different features of computation — parallelism, side-effects, non-determinism, polymorphism, higher-order functions, etc. Some of these features are still not adequately understood on the algebraic level, even when considered in isolation from other features. It makes sense to develop complex logics from simpler logics in a structured fashion, just as complex specifications are built in a structured fashion from simpler units. Methods for building complex logics in this fashion are only beginning to be explored.

Problem Development of algebraically-based logics for dealing with more features of computation, and of methods for building complex heterogeneous logics from simple logics in a structured fashion.

The development of programs from specifications takes place via a sequence of implementation steps. Between each step (or during steps, depending on the formalism in question) the specification may be decomposed into a number of smaller specifications which are then implemented separately. This gives rise to a tree of implementation steps with program fragments at the leaves. Vertical and horizontal composability theorems guarantee that if all the individual implementation steps are correct then the program which results from an appropriate combination of all these program fragments will be a correct implementation of the original specification.

The basic foundations of this style of program development have been well-studied. But so far this technology has not been applied much except to toy examples. There are at least two reasons for this. First of all, it is difficult to come up with implementation steps. Schematic transformation rules which are guaranteed to produce correct implementation steps are useful at some stages of the implementation process, but heuristic methods which hint at reasonable directions in which to proceed are also needed. This is the main creative step in program development and so no complete solution is to be expected, but the topic is worthy of investigation. It is quite possible that such methods will be specific to problems coming from certain subject areas (e.g. databases, text manipulation, etc.). Second, it is difficult to prove the correctness of implementation steps once they have been proposed. Proofs are especially difficult when behavioural equivalence is involved. Both of these topics need further work.

Problem Methods (possibly heuristic methods) which make it easier to come up with implementation steps.

Problem Methods for proving the correctness of implementation steps, especially when behavioural equivalence is involved.

Another method for developing programs from specifications is via the "proofs as programs" paradigm whereby a constructive proof of a theorem of the form $\forall X.\exists Y.R(X,Y)$ gives rise to a program which takes X as input and produces Y as output such that $R(X,Y)$ holds. The relationship between these two methods needs to be understood better. If an appropriate relationship can be established then benefits should flow in both directions.

One of the great benefits of formally developing software from specifications is the possibility of reusing some program modules in later projects. One may imagine formal program development becoming almost practically feasible in spite of its great unitary cost because of the potential of spreading this cost over many different projects. It is therefore very important to know under which conditions implementations of specifications may be reused. This seems like an important question but it is not a problem which needs to be solved, if the program development framework is set up properly. Suppose that the specification SP is implemented by the program module P. If SP recurs as the specification of a module in the design specification of another project, then P is reusable by definition (or else there is something seriously wrong with the program development framework). If SP recurs as part of a requirements specification, then it is premature to worry about implementation and so the question of reusing P does not arise.

Non-problem Conditions under which implementations of specifications may be reused.

The eventual practical feasibility of formal program development hinges on the availability of computer-aided tools to support various development activities. This is necessary both because of the sheer amount of (mostly clerical) work involved and because of the need to avoid the possibility of human error. Existing systems tend to concentrate on one or two aspects of the specification and formal development process. It is clear that an integrated environment to support formal development would include a wide variety of tools. Much more work is needed here. At this stage it is even difficult to say definitely what is needed.

Problem Development of tools to support specification and formal development.

A number of support tools have already been developed, but so far the main emphasis has been on making specifications executable using term rewriting which we have argued above is a non-problem. There is definitely a role for term rewriting in a support system for formal program development, but it seems to us that this is a subtopic of the much more general problem of theorem proving. One topic which has been widely ignored is how to prove theorems about structured specifications. The structure of specifications introduces extra problems, since structured specifications cannot always be reduced to flat specifications. On the other hand, there is some indication that the structure of specifications can be helpful in guiding proof search.

Problem Theorem proving in structured specifications.

One would hope that institution-independent foundations for specification and formal development would be reflected at the level of support tools. The practicalities of this are very unclear, although some progress has already been made in the area of theorem proving.

Problem Translating institution-independence of foundations to the level of support tools.

Another problem is the way in which such a collection of tools is integrated to form an environment. There are many issues here, including such things as the choice of an appropriate user interface and how tightly the components of a support system should be coupled.

Problem Architecture of environments to support formal program development.

3 Conclusion

The above list of problems and non-problems reflects our current state of thinking about these topics. When something is listed as a non-problem, this is not to be interpreted as an assertion that it is uninteresting or not worth studying; it is merely a statement that we do not see how "solving" it will contribute much to progress towards the ultimate goal of formally developing programs from specifications. We hope that people who do not share our view will take this is as an invitation to explain their motivation with respect to this goal, or with respect to some other explicitly-stated ultimate goal.

Acknowledgements

The ideas in this note were presented by DS during a panel discussion at a meeting of the COMPASS working group in Bremen in October 1989. The other panelists were Egidio Astesiano, Michel Bidoit, Hans-Dieter Ehrich, Bernd Krieg-Brückner, and Fernando Orejas. We thank these people and the audience for the ensuing discussion which suggested ways of refining the presentation herein. Thanks also to Stefan Sokołowski for comments on a draft. DS received travel support from the COMPASS working group which is funded by ESPRIT.