Interfaces and Extended ML

Stefan Kahrs* Donald Sannella† Andrzej Tarlecki‡

Abstract

This is a position paper giving our views on the uses and makeup of module interfaces. The position espoused is inspired by our work on the Extended ML (EML) formal software development framework and by ideas in the algebraic foundations of specification and formal development. The present state of interfaces in EML is outlined and set in the context of plans for a more general EML-like framework with axioms in interfaces taken from an arbitrary logical system formulated as an *institution*. Some more speculative plans are sketched concerning the simultaneous use of multiple institutions in specification and development.

1 Interfaces in general

Modularisation mechanisms in programming languages such as C++ [Str86] and Standard ML (SML) [MacQ86] provide useful tools for coping with the complexity inherent in large software systems. A central ingredient of such schemes is the use of *interfaces* to mediate module interconnection. A module interface is a description of the facilities that the module makes available for use by the rest of the system. The amount of detail recorded in this description is generally less than that of the implementation provided by the module body; it glosses over (i.e. abstracts away from) some of the arbitrary choices made in the implementation. This loss of information serves at least two purposes, corresponding to two vantage points: that of the other modules of the system; and that of the module itself. From the former point of view, the interface highlights the essential features of the module rather than burying them within a morass of unimportant details. From the latter point of view, omitting information that should be of concern only to the implementor of the module enables implementation details to be changed later without affecting the rest of

*smk@dcs.ed.ac.uk; Laboratory for Foundations of Computer Science, Edinburgh University. This research was supported by SERC grant GR/J07303.

†dts@dcs.ed.ac.uk; Laboratory for Foundations of Computer Science, Edinburgh University. This research was supported by SERC grants GR/J07303 and GR/J07693, a SERC Advanced Fellowship, and the COMPASS Basic Research working group.

‡tarlecki@mimuw.edu.pl; Institute of Informatics, Warsaw University, and Institute of Computer Science, Polish Academy of Sciences. This research was supported by SERC grant GR/H76739, an EC-funded COST fellowship, and KBN grant 2 P301 007 04.

the system. These are two sides of one coin: the interface defines what the module can be depended on to provide, without constraining the means used to provide it.

In a programming language, interfaces record the names and (usually) types of module components. This is exactly the information about a module required for the (separate) compilation of subsequent modules that depend on it. However, this "static" information is not sufficient when the objective is proving correctness of a modular system with respect to some specification of its required behaviour; in this case, it is necessary to add "logical" information about the properties of module components. This, in turn, is exactly the information about a module required for the (separate) verification of subsequent modules that depend on it.

Interfaces containing such logical information play a central role in frameworks for specification and formal development of modular systems such as Extended ML (EML) [ST85]. Formal development of a module involves proceeding from such an interface to a module body that is a provably correct implementation of the interface. Here, interfaces mediate proofs of correctness; a module is proved to correctly implement its interface on the basis of those properties of modules on which it depends that are recorded in their interfaces. As a result, it is possible to prove that a module is correct even before the modules on which it depends have been fully implemented. This enables work on the development of modules of a large system to be carried out top-down (or "inside-out") rather than bottom-up, and enables work to proceed simultaneously on related modules without danger of conflict. In order for interfaces to be of much use in formal development and verification, they have to have a formally defined meaning; otherwise proofs of properties of modules are out of the question.

This is a position paper setting out our views on the uses and makeup of module interfaces. Section 2 outlines the syntax, semantics and role of interfaces in the EML formal software development framework. Section 3 describes how the present state of EML relates to our plans for a general EML-like framework with axioms in interfaces taken from an arbitrary logical system; the semantic basis for this is Goguen and Burstall's concept of *institution*. Section 4 concludes with a sketch of some more speculative ideas concerning the simultaneous use of multiple logical systems in the specification and development of "multi-paradigm" systems built from heterogeneous components, and of ordinary "uni-paradigm" systems.

2 Interfaces in Extended ML

EML is a framework for the formal development of modular SML software systems from specifications of their required behaviour. The long-term goal of work on EML is to provide a practical framework for formal development together with an integrated suite of computer-based specification and development support tools and complete mathematical foundations to substantiate claims of correctness. Although considerable progress has been made, this goal is still a long way off; see [ST89], [ST91], [San91], [KST93a] and [KST93b] for the details that are omitted in the brief and very informal presentation below.

The EML specification language is a simple extension of SML whereby *axioms* are permitted both in module interfaces to specify the properties of module components, and in place of SML code in module bodies. As in SML, ordinary non-parameterised modules are called *structures*, and parameterised modules (taking structures as parameters) are called *functors*. Probably the most commonly-cited example of a functor is a package for sorting lists containing values of an arbitrary type with respect to some arbitrary order relation on values of that type; here, the parameter defines the particular type and order relation of interest, and application of the functor to that parameter yields the required sorting program. A structure has a single interface (called a *signature*) specifying its components, while a functor has both an input signature to specify requirements on permissible structure parameters, and an output signature to specify the components of the structure which results when the functor is applied. In contrast to SML, interfaces in EML are *opaque*, meaning that only the information recorded in a module's interface (or interfaces, in the case of a functor) is available externally. With *transparent* interfaces as in SML, information about the representation of type components of a module can be exploited by subsequent modules; this is sometimes convenient, but it has the very unfortunate consequence that changing to a different representation may cause code in other modules to stop working.

In SML, when one module (structure or functor) uses components from another module, when a functor is applied to a structure, or when a module is declared as having a given signature, the system automatically checks for type compatibility. The main mechanism here is that of *signature matching*, i.e. comparing interfaces to ensure that what is required is in fact supplied. The same goes for EML, except that signature matching has to be extended to take account of axioms in signatures as well. The language of EML axioms (see below) is far too powerful to enable such checks to be carried out automatically, so signature matching gives rise to *proof obligations* which need to be discharged (i.e. the proofs need to be carried out) in order to guarantee compatibility [San93].

Axioms specify the functional behaviour of module components, in the tradition of the algebraic specification paradigm. Any expression of type `bool` may be used as an axiom, which amounts to an assertion that the expression evaluates to the value `true`; that is, the built-in datatype `bool` is identified with the type of logical values in the logic. The basic logical connectives are those of SML (`andalso`, `orelse`, `not`) with the additional connective `implies`. Universal and existential quantification is provided over all types, including function types and polymorphic types. A "logical" equality predicate `==`, which can be used to compare values of any type, complements the "computational" equality `=` provided by SML which can only be used for values of a so-called *equality type*. Logical equality is extensional equality on function types as well as with respect to exceptions and non-termination: *exp*`==`*exp* is `true` even if *exp* raises an exception or fails to terminate. Two additional predicates are provided: one tests if evaluation of an expression terminates or not, and the other tests if evaluation raises an exception. The design of a language of axioms that is rich enough to cope with SML raises a number of interesting technical problems; see [KST93a] for relevant discussion.

EML covers all of SML, with the exception of references (pointers), but including polymorphic types, non-terminating computations, exceptions, user-defined types and higher-order functions. References are omitted for the sake of simplicity, but it would not be too difficult to treat them once it is decided what the existing logical constructs should mean in the presence of side effects. For example, should exp==exp' mean just that the values of exp and exp' are identical, or does it also require the side effects of evaluating exp and of evaluating exp' to coincide?

Formal development of a software system from a specification of requirements (consisting of a single signature in the case of a structure, or pair of signatures in the case of a functor) proceeds top-down by stepwise refinement and modular decomposition. In the latter, the problem is decomposed into a number of simpler problems by specifying a number of new modules and defining the module at hand as a composition of these. Providing a body for each of these new modules is a self-contained task; these tasks can be tackled separately in any order, precisely because the signature(s) of each required module defines exactly what it needs to know about the outside world and what the outside world requires it to do.

3 Extended ML in an arbitrary institution

The basic ideas underlying EML do not actually depend on the particular features of the SML language or of the logical notation used to write axioms. What *is* essential is SML's module system (with the use of opaque interfaces as described above): the concepts of signature, structure and functor, and the manner in which they can be defined and used. This module system can be adapted for use with a wide variety of programming languages; for example, see [SW92] for an SML-inspired module system for Prolog. In a similar way, it is possible to adapt EML for use with different programming languages. Even in a given programming language, it is possible to use different logical systems for writing axioms describing the behaviour of module components. In early work on EML ([ST85], [ST86], [ST89]), we explicitly aimed for a framework with this degree of flexibility. More recently, we have been concentrating on the specific case of SML and the language of axioms sketched in Section 2, but the foundations underlying the EML formal development methodology support much more than this special case.

The semantic basis for this flexibility is Goguen and Burstall's concept of an *institution* [GB92], which is a particular formulation of the intuitive concept of a logical system. In simple terms, an institution *INS* comprises a notion of signature and, for each signature, a collection of semantic models over that signature, a collection of well-formed axioms over that signature, and a satisfaction relation defining which models satisfy which axioms. This gives a basic framework in which axioms can be used to specify classes of models; when the models correspond to software modules, such a specification amounts to a description of the permissible implementations of a module interface. In the institution appropriate for the EML framework as described in the previous section, semantic models correspond to structures in SML and signatures correspond to signatures in SML. The language of axioms and what it means for a model to satisfy an axiom is as described earlier. An institution similar to the one required is defined in [Kaz92]. Given an arbitrary institution *INS*, an EML signature amounts to a signature of *INS* together with a set of axioms of *INS* that are well-formed in the context of that signature, and the meaning of the axioms is given by the satisfaction relation of INS. See [ST86] for a sketch of a semantics for EML in the context of an arbitrary institution, and see [ST89] for a justification of the soundness of the EML formal development methodology that is applicable in a similar setting.

Instantiating EML to give a specification and formal development framework for a given programming language requires first that an SML-like module system be added to the language, and then that an institution be formally defined having signatures corresponding to signatures of modules, semantic models corresponding to structures, and axioms appropriate for specifying the components of such structures, with the meaning of axioms defined by the satisfaction relation. Not all of the features of the SML module system need to be present for this to work; for instance, EML-style formal development still works in a module system lacking the concept of a functor.

None of this makes sense in the absence of a formal semantics for the programming language at hand (for SML, see [MTH90] and [MT91]): it must be completely clear exactly which structures (containing code written in that language) correspond to which models of the institution. Furthermore, defining a language of axioms in institutional terms involves defining exactly when models satisfy axioms; again, the key is a formal definition. But note that the language of axioms is unconstrained; in particular, it is *not* restricted to assertions about functional behaviour. Axioms of any kind are permissible, provided that it is possible to give an unambiguous definition of when a structure satisfies an axiom. So for example, this approach encompasses interface specifications containing efficiency constraints, since it is possible (in principle at least) to spell out exactly when such a constraint is satisfied. On the other hand, it probably does not encompass interface specifications containing the requirement that a module be "maintainable" or "reliable"; this is not because of any philosophical beliefs concerning the usefulness of such specifications, but because it is difficult to see how to give a reasonable definition of exactly when such a constraint is satisfied.

Although the foundations underlying EML happen to be based on institutions, the same points would apply if they were based on some other formulation of the intuitive concept of "logical system", including both algebraic-style competitors to institutions (e.g. [Poi89], [EBO93], [SS93]) and type-theoretic formulations like the Edinburgh Logical Framework [HHP93]. Our point is that the definition of the logical system used must be explicit and the correspondence between programs and this logical system must be clear; beyond this, anything goes as far as we are concerned. Parameterizing a specification framework by an arbitrary institution gives the ability to use different logical systems and different programming languages in the same framework without the need to re-build it from scratch.

4 Extended ML in multiple related institutions

The possibility of using a single specification and formal development framework with different institutions has been mentioned above. But even in the process of developing a single software system it may be convenient to use different institutions at different stages of development. After all, we proceed from a high-level user-oriented specification to low-level computer-oriented code; it seems only natural that different logical tools are necessary to express properties at these very different levels. Another reason why we might want to use multiple institutions in the construction of a single system is in the case of so-called *multi-paradigm* systems built from heterogeneous components. For example, a different institution would be suitable for specifying and reasoning about a concurrent subsystem (say, Hennessy-Milner logic [HM85]) than for developing a module implemented using a logic programming language (say, first-order equational logic). This includes also the development of mixed hardware/software systems, which would involve the use of an institution suitable for hardware description (say, higher-order logic [Gor86]).

When multiple institutions are applied in the construction of a single system, some way of relating the institutions to each other is required. There are several ways of relating institutions; the one that seems most relevant for this purpose is the concept of *institution semi-morphism* [ST94] (cf. *semi-institution morphisms* in [ST88]). Informally, an institution semi-morphism maps the models of one institution to those of another; the direction of the map is from the "richer", more detailed institution to the "poorer", less detailed and hence more abstract one. This model translation map may be thought of as a projection function which strips away aspects of the model that are irrelevant in the poorer institution. Since the signatures provided by the two institutions may differ, the model translation map is accompanied by a translation of signatures going in the same direction. No relation between the axioms of the two institutions nor between their satisfaction relations is required; this is the reason why this is called an institution *semi*-morphism. (The concept of *institution morphism* in [GB92], which also includes a translation of axioms, is too restrictive for use in the present context.) One may wish to view institution semi-morphisms as interfaces between logical systems, but that is a topic for a different paper!

When different institutions, related by institution semi-morphisms, are used in the development of a single system, the various model translation maps are used to make sense of the relationship between descriptions of the same module at different stages of development, and to mediate interconnections involving modules of different kinds. An interesting aspect of the latter is that a model translation map can serve to hide irrelevant details of a module implementation which are an artefact of the paradigm used. For example, the communications between components of a concurrent subsystem are hidden from view when all we are interested in is regarding it as a particular way of implementing a collection of functions.

The ideas sketched in this final section are speculative and somewhat half-baked. The foundations exist, but little thought has gone into putting them into practical use. In particular, we have not yet considered how these ideas may be incorporated into a concrete EML-like framework for specification and formal development.

References

[EBO93] H. Ehrig, M. Baldamus and F. Orejas. New concepts of amalgamation and extension for a general theory of specifications. *Proc. 8th Workshop on Specification of Abstract Data Types*, Dourdan. Springer LNCS 655, 199–221 (1993).

[GB92] J. Goguen and R. Burstall. Institutions: abstract model theory for specification and programming. *Journal of the Association for Computing Machinery* 39:95–146 (1992).

[Gor86] M. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. *Formal Aspects of VLSI Design* (G. Milne and P.A. Subrahmanyam, eds.). North Holland (1986).

[HHP93] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery* 40:143–184 (1993).

[HM85] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the Association for Computing Machinery* 32:137–161 (1985).

[KST93a] S. Kahrs, D. Sannella and A. Tarlecki. The semantics of Extended ML: a gentle introduction. *Proc. Intl. Workshop on Semantics of Specification Languages*, Utrecht. Springer Workshops in Computing, to appear (1993).

[KST93b] S. Kahrs, D. Sannella and A. Tarlecki. The definition of Extended ML. Draft report, Univ. of Edinburgh (1993).

[Kaz92] E. Kazmierczak. Model theory for Extended ML. Draft report, Univ. of Edinburgh (1992).

[MacQ86] D. MacQueen. Modules for Standard ML. In: Report ECS-LFCS-86-2, Univ. of Edinburgh (1986).

[MT91] R. Milner and M. Tofte. *Commentary on Standard ML*. MIT Press (1991).

[MTH90] R. Milner, M. Tofte and R. Harper. *The Definition of Standard ML*. MIT Press (1990).

[Poi89] A. Poigné. Foundations are rich institutions, but institutions are poor foundations. *Proc. Intl. Workshop on Categorical Methods in Computer Science with Aspects from Topology*, Berlin. Springer LNCS 393, 82–101 (1989).

[SS93] A. Salibra and G. Scollo. A soft stairway to institutions. *Proc. 8th Workshop on Specification of Abstract Data Types*, Dourdan. Springer LNCS 655, 310–329 (1993).

[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).

[San93] D. Sannella. Static and logical correctness conditions in formal development of modular programs. Draft report, Univ. of Edinburgh (1993).

[ST85] D. Sannella and A. Tarlecki. Program specification and development in Standard ML. *Proc. 12th ACM Symp. on the Principles of Programming Languages*, 67–77 (1985).

[ST86] D. Sannella and A. Tarlecki. Extended ML: an institution-independent framework for formal program development. *Proc. Workshop on Category Theory and Computer Programming*, Guildford. Springer LNCS 240, 364–389 (1986).

[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. Joint Conf. on Theory and Practice of Software Development*, Barcelona. Springer LNCS 352, 375–389 (1989).

[ST91] D. Sannella and A. Tarlecki. Extended ML: past, present and future. *Proc. 7th Workshop on Specification of Abstract Data Types*, Wusterhausen. Springer LNCS 534, 297–322 (1991).

[ST94] D. Sannella and A. Tarlecki. *Foundations of Algebraic Specifications and Formal Program Development*. Cambridge Univ. Press, to appear (1994).

[SW92] D. Sannella and L. Wallen. A calculus for the construction of modular Prolog programs. *Journal of Logic Programming* 12:147–177 (1992).

[Str86] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley (1986).