April 1991

Breitenbush and Santiam : tools for program development in Extended ML

James Hook

Follow this and additional works at: http://digitalcommons.ohsu.edu/csetech

Recommended Citation
http://digitalcommons.ohsu.edu/csetech/237

This Article is brought to you for free and open access by OHSU Digital Commons. It has been accepted for inclusion in CSETech by an authorized administrator of OHSU Digital Commons. For more information, please contact champieu@ohsu.edu.
Breitenbush and Santiam:
Tools for Program Development in Extended ML

James Hook

Oregon Graduate Institute
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 91-010

April, 1991
Abstract

We propose to design and implement an environment for computer assisted program development in Extended ML, a formal specification language. We intend to demonstrate the feasibility of the EML methodology, to demonstrate its applicability to a large class of problems, and to explore issues that arise in providing an automated programming environment for formal methods. We will produce an EML derivation editor (Breitenbush) built with the Cornell Synthesizer generator, a library of examples and technical reports describing our experiences, and detailed plans for a second-generation environment (Santiam).

1 A personal vision

In the 1990s and beyond increasingly complex problems involving ever more critical systems will have to be solved by fewer software engineers than we have today. Yet, we are already straining the limits of our current software development technology—it cannot scale much further. The AT&T telephone network has exhibited a catastrophic software failure; companies regularly fail to deliver software on schedule; and the accepted standard for safety-critical systems is still informal review and extensive testing.

To meet the needs of the future, we must improve the productivity of software engineers by reducing the number of intellectual acts they perform each day while increasing the quality of the software they produce. We believe that this revolution in programming will come by the application of formal methods to software development.

Formal methods will support reasoning at appropriate levels of abstraction—decreasing the number of bookkeeping and representational details that a programmer must manage.
while increasing confidence in the result. In the end, the programmer must say more,
and say it more clearly, by writing fewer lines of code.

We envision the formal design of a system as an active entity maintained by the
software development environment. Early in the design phase the design specification
can be explored by proving properties of the integrated components based on their spec-
ifications. As the design evolves, some simple tasks may be prototyped by automatic
programming techniques, possibly based on program extraction from constructive proofs
as in Nuprl and Romulus[7, 10], or on AI heuristics such as in KIDS[48]. As parts of
the system are implemented, the proof of compliance with the specification will be main-
tained. If, in the future, design changes are made that invalidate correctness proofs of
system components, the environment will automatically discover the invalidated claims
and bring them to the attention of the software engineer.

The discipline of formal methods is also expected to assist the programmer by aiding
in clear thinking—the most important contribution of a team member to any system.
Companies currently are experiencing improved communication between engineers and
higher productivity through the use of rigorous methods[9]. Rigorous methods are meth-
ods in which designs are expressed in a mathematical notation but, in contrast with
formal methods, implementations are not formally proved to correspond to designs.

While advocating discipline as contributing to clear thinking, we do not deny that
programming and algorithmic design is fundamentally a creative process. It is important
that the software development environment of the future support a sufficiently expressive
logic to facilitate the development of efficient and creative algorithms. This includes being
expressive over algorithms involving state and control.

If this is the future, how do we get there? How long will it take? We feel that
research into formal methods, logical frameworks and programming language design is
now at a point where systems realizing important facets of this vision are a poten-
tial reality. We propose to build a system for program development based on Sannella
and Tarlecki's Extended ML specification logic and the associated program development
methodology[41, 42, 39, 44]. The system will provide support for a single user to manip-
ulate an integrated program-design object, giving automatic assistance for the stepwise
development of programs from specifications.

This system will not address all the problems outlined above. In particular, we will
not address the problems of concurrent access, version management, automatic program-
ning and theorem proving in this effort. However, it will initiate a long-term research
program that will address those issues in the future.
2 Why EML?

The environment of the future will have a specification language at its core. The specification language will properly include an executable programming language, as well as a mathematical logic expressive over the abstractions present in the design. In this project we will take Standard ML (SML) as our implementation language and an augmentation of it called Extended ML (EML) as our specification language.

Standard ML is a small, formally defined programming language with higher-order functions, references, exceptions and strong-typing[23]. It comes out of a tradition of languages based on the lambda-calculus that goes back to Landin’s ISWIM, introduced in his classic 1966 paper on the fundamentals of programming languages[19]. Standard ML is arguably the most mathematically elegant language designed and implemented to date that includes both functional and imperative features. It is an obvious choice as a starting point for this investigation.

Standard ML includes a module facility that supports the data encapsulation techniques popularized in object-oriented programming, while retaining static-typing and strong-typing. The basic unit of data encapsulation in SML is called a structure. Like an “object,” a structure is a collection of types and values. The type of a structure is called a signature. Signatures record the visible interfaces of structures, much like class definitions describe objects. However, signatures and structures are completely distinct entities, and neither are first-class values at runtime, unlike objects and classes. Instead structures are manipulated at compile time, where they are acted upon by special functions called functors. The functor-structure calculus provided at compile time is rich enough to express much of the code reuse claimed by advocates of object-oriented methods. (This mechanism is also similar to, but far more general than, Ada’s generic packages.)

In spite of this immense expressive power, the semantic explanation of SML is compact and mathematically well-behaved[16, 17, 24]. Object-oriented languages, in contrast, have complex and quite operational semantic explanations. Formal reasoning about even the simplest object-oriented language is extremely difficult.

Extended ML is a family of specification logics built on top of Standard ML. It has been defined by Sannella and Tarlecki of Edinburgh and Warsaw, respectively. All logics in the EML family support the same development methodology and have the same high level structure. They differ, however, in exactly which features of Standard ML they can naturally express. It is not surprising that the simplest logics deal with the most restricted subset of the language. One area of investigation will be finding the appropriate balance between the expressive power of the logic and its over-all simplicity.

Theoretical work on EML began in 1985 and continues today[41, 42, 39, 44]. There is currently an implementation project at Edinburgh that is focusing on building compiler-like tools for EML that interface to theorem proving engines being developed with sup-
port from the UK Science and Engineering Research Council[40]. We are in close communication with the principal investigators of these efforts. We know of no other efforts investigating EML in the United States. More information about EML is given in the appendix.

EML has also been used in an industrial setting by Harlequin Ltd of Cambridge, England. They have use it for specification of a compiler, and plan to specify other components of their comprehensive Standard ML design and development environment[47].

3 Breitenbush—a derivation editor

The Breitenbush derivation editor will be the first tool we build in support of our vision of the future. It will directly support the EML methodology for deriving programs that satisfy specifications. This methodology is based on decomposition and refinement. The editor will augment this paradigm with a mechanism of design reuse, discussed below.

The editor will realize the vision of a computer-maintained entity representing the design and implementation. This entity will be the derivation of the program from its specification. It will include the top-level specification and be built by the three basic steps of decomposition, coding and refinement provided by the EML methodology.

The EML derivation will be viewed as a tree. The root of the tree gives the highest level specification of the problem. The branches in the tree reflect design decisions that result in decomposition or refinement. The leaves of a complete derivation yield an executable SML program that satisfies the specification in the root. Every well-formed derivation may be read as a consistent story of how the implementation relates to its specification.

The primary operations supported by the editor will correspond to the basic EML derivation operations: decomposition, coding and refinement. The user will select a leaf of the derivation tree to be extended, and invoke the appropriate derivation operation. The system will then transform the derivation, grafting a new subderivation onto the tree. The system will also calculate the logical conditions that must be demonstrated to justify the derivation step. These correctness conditions are a critical part of the derivation. Breitenbush will provide limited support for these proofs.

When working out derivations by hand, there is a great deal of information that is repeated from one stage to the next, and it is difficult to keep track of the implications of changing an earlier design decision. To facilitate this, and to rapidly get a mature, workstation-based user interface, we plan to use the Cornell Synthesizer Generator[34, 36, 37] to build the Breitenbush editor. The synthesizer generator is a tool that builds editors from language descriptions given by attribute grammars. Attribute grammars provide a way to augment context free grammars with context sensitive information, expressed as equational constraints between “inherited” and “synthesized” attributes.
The synthesizer has been used successfully to prototype a user interface for a logical framework based proof editor[12] and an Ada verification environment[14]. It will allow us to build a demonstration project quickly.

The attribute propagation mechanism provided by the editor transmits the effect of a change in a component throughout the tree, providing a tool akin to a spread-sheet calculator for derivations. We expect this feature to provide a design reuse mechanism through which a user will load a previously developed derivation into the current derivation and update its specification as required. (If no update is required this is traditional code reuse, which is also supported.) The synthesizer will then propagate the impact of these changes throughout the derivation and identify places that need further attention. We feel that design reuse is the “good idea” behind the overly powerful and potentially dangerous notion of method inheritance in object-oriented programming. An example illustrating EML and design reuse is given in the appendix.

Once the editor is complete we will work on a series of examples that demonstrate:

1. the utility of the EML methodology,
2. the enhanced utility of an automated EML system,
3. the application of EML to several different problem domains,
4. design and derivation reuse,
5. the design of specifications to promote reuse, and,
6. the simplicity and learnability of the EML paradigm.

In addition we will determine how appropriate EML is for the larger OGI effort on formal methods for software development. We expect this project to support directly a larger effort proposed by Kieburtz, Hook and Bellegarde on deriving software from specifications.

4 Santiam—the next step

While we anticipate learning a great deal from Breitenbush, it will not by itself become the tool to realize our vision of the future. The synthesizer generator will allow us to develop a very nice user interface with minimal effort, but it will not communicate directly with the tools supporting formal methods that currently exist in the research community. Nor is it well suited for algorithmically extensible environments, such as those provided by the tactic mechanism in LCF. In the long term, we expect the automatic programmer's assistant, briefly described in the first section, to be supported by a tactic-like
mechanism. For these reasons we include in the goals of this initiation project the design and exploratory implementation of the next generation system: Santiam.

The Santiam system will be developed in Extended ML and implemented in Standard ML. It will not have the benefit of the synthesizer's built-in attribute recalculation engine. We plan to use its design and implementation as a test case for the EML methodology in general and the Breitenbush editor in particular. It will provide support for automating derivation methods discovered in the Breitenbush system, automatic programming using user-supplied program search tactics, and formal verification of the soundness of logical derivations.

4.1 Tactics

The notion of tactic was introduced by Milner in the Edinburgh LCF system[11]. Naively, a tactic interface is very much like a command interpreter, such as the UNIX shell. At the interactive interface, there is no distinction between built-in commands, system supplied utilities and user programs; shell commands can be combined in a simple programming language to form other shell commands; and, most importantly, the observable effects of a shell command must be exclusively realized through system calls. Ultimately, there is only this small distinguished class of primitive actions that can be effected by a command. Similarly, in a tactic based theorem prover, tactics may be invoked without knowing if they are primitive rules or user defined ones; they can be combined and manipulated by programming in the metalanguage; and they are constrained to use only a small set of primitive rules (the inference rules of the logic).

Tactics are more complicated than shell commands, however, because it is necessary for the composition of their results to form complete derivations that accomplish specific goals. Typically, the goal is a theorem and its accomplishment is a formal proof. To enforce these constraints, tactics are written as programs in a strongly-typed programming language. Each tactic is a function from the desired goal to a list of subgoals and an object called the validation. If tactic \( t \) applied to goal \( G \) gives subgoals \( G_1 \) and \( G_2 \), that means if goals \( G_1 \) and \( G_2 \) are satisfied then their solutions can be combined to solve \( G \) using steps ultimately justified by primitive rules. This is done by applying the validation, which is a function, to the accomplishments of \( G_1 \) and \( G_2 \) to yield the accomplishment of \( G \).

Tactics provide a rich paradigm for goal-directed programming. Tactics may either implement derived logical rules or heuristic search. Tactics can be supported by refinement style editors, as in Nuprl[8], where proofs are represented by trees. In such systems, tactics are associated with nodes in the tree and justify the relationship between a node and its children. Tactics may also return proof objects or extracted code obtained from the proofs they discover. An abstract treatment of tactics is presented in Griffin's thesis[13].
To integrate tactics into Santiam, it will be necessary to identify how goals and their validations are to be expressed. It is expected that at one level an EML specification will be a goal and a complete derivation yielding SML code will be the record of its accomplishment. However, there will be other notions of goal and validation at other levels of granularity, such as theorems and extracted proofs.

4.2 Tactics and information flow

A major question which must be addressed in the implementation of Santiam is how much of the information propagation style of Breitenbush can be retained in a system that supports tactics. In Breitenbush, the information flow is specified with first-order attribute equations and the information propagation is done by the synthesizer's incremental attribute evaluation algorithm. Allowing an extensible tactic collection introduces higher-order dependence. This makes the static analysis techniques of the synthesizer generator inapplicable.

This general problem of incremental change in an environment based on tactics arises in several similar contexts, including LCF, Nuprl, and Romulus. The work of Bundy[4, 5] Paulson[29] and others will be relevant to this investigation.

Although our initial goal in building Santiam will be to facilitate the derivation process from specification to program, it should also, in principle, provide an environment for more traditional program transformation. In Santiam's design we will examine tools implementing these techniques. In particular, we will study the Orme tool set[20] and the Focus system[33].

4.3 Theorem proving support

In addition to providing an extensible derivation environment, Santiam will also have access to theorem proving tools implemented in Standard ML, such as Paulson's generic theorem prover Isabelle[30] and Pollack's logical framework implementation LEGO[32, 21]. This link will be vital to make the system formal rather than simply rigorous. In the truly formal setting it is important that the logic be easily changed to support the research outlined in the section on theoretical investigations below.

Some of the concepts in Santiam are also being explored at the University of Edinburgh. We will cooperate with Sannella's group; we intend to avoid unnecessary duplication of effort.

The Santiam system is considerably broader in its overall scope than Breitenbush. Experience with Breitenbush will determine the priorities assigned to the various aspects of Santiam. We will flesh out its design and initiate its implementation in the second year of this project. Its completion, however, is beyond the scope of this proposal.
5 Theoretical investigations—a continuing effort

Not all of the work on the environment of the future is centered around implementation—there is still a lot of theory to be worked out with pencil and paper. As the field of computer science matures, we are using more and more advanced mathematics to discover and express the fundamental simplicity of computational processes. This paradoxical situation holds in specification logics and programming language design, where new advances are being made by organizing logical and computational notions in category theory, an abstract theory of functions and function spaces originally developed by topologists. Recent advances in language semantics that use category theory as an organizational tool appear very attractive.

5.1 How wide a spectrum?

Sannella and Tarlecki describe EML as a wide spectrum language because it contains both a specification logic and an implementation language. When that implementation language is full Standard ML then EML is clearly wide spectrum. However, when SML is first restricted to the first-order recursion equations used in the motivational papers, the width of the spectrum is considerably reduced.

This narrower spectrum language, while too restrictive for some problems, is certainly appropriate for many interesting examples. When it is applicable, it is often much simpler than more expressive logics. It is desirable to allow the user to use the simpler logic of the narrower spectrum language on some components of a derivation and a more wide spectrum language on others.

For example, suppose the sort example developed in the appendix is to be used in a program that reads an input stream and writes an output stream. It would be best to decompose this into three components: input, sort and output. The input component could be reasoned about in a logic that captured the semantics of interactive input, the sort component could be derived in the simple equational setting, and the output component could be developed in a logic that expressed interactive output. The entire program could then be integrated in a single logic of interactive input and output that extended all three of the logics used in the derivation.

Furthermore, after the sort implementation was derived in the elementary logic of recursive equations, it may be desirable to refine it into a functionally equivalent implementation using stores and iteration. This derivation would require a logic of sequential computation with modifiable stores. It also requires a theoretical justification of the use of the imperative implementation in place of the original functional program. This may be obtained by adapting results of Sannella and Tarlecki on implementations of algebraic specifications to this context[43].

We will investigate a “multi-spectrum” approach where different components may be
developed in different logics, as appropriate, and then combined in a sound manner. We are encouraged by Moggi's work on modular denotational semantics[26, 27]. He shows how to combine different semantic facets in a categorical setting to get denotational semantics for languages in a systematic way. However, it is not clear to what extent his constructions preserve interesting theorems in the logic.

Moggi and Pitts are also applying this machinery to what they call evaluation logics, which are logics generated by categorical computational models[25, 31]. These logics can express modalities, such as those found in the dynamic logics used in concurrency.

If the multi-spectrum approach can be given a sound foundation its utility must still be demonstrated. We will design Santiam to accommodate multiple logics. This will result in a testbed for the multi-spectrum approach. We will also study carefully the EML logics developed by Sannella and Tarlecki. They are currently working on a logic that includes exceptions and higher-order functions.

5.2 Alternate foundations
We are also looking at alternative foundations for EML. In a recent visit, Robert Harper suggested that a type-theoretic foundation may be given to the EML specification language. This move from a model-theoretic semantics to a type-theoretic one may simplify the system and integrate more naturally with other semantics accounts of ML. In particular, the treatment of polymorphism may be more natural in the alternative framework, which is based on Harper’s work with Mitchell and Moggi on higher-order modules[17]. Sannella and Tarlecki’s work on higher-order specifications may also be relevant to this[45].

6 Related work
The EML methodology supports what Scherlis and Scott call inferential programming[46]. It does this in the context of a wide spectrum language. The Munich project CIP is the most complete example of an automated environment supporting this approach to program development. The Munich group defined a wide spectrum language, CIP-L[2], and implemented a derivation environment, CIP-S[3].

The language CIP-L differs from EML in two very important ways: it attempts to be more inclusive of low level features, such as concurrency and non-determinism, and it has less support for the modular development of programs.

As a member of the Standard ML committee, Sannella made sure that the support for programming in the large in SML would be appropriate as the basis of a system for formal reasoning. Consequently, EML appears to be a better base for inferential programming than CIP-L, particularly because of its connection with the Standard ML
module facility.

The CIP-S system supports inferential programming by maintaining complete derivations. It does not, however, provide the support for identifying the scope of changes that supports our design reuse paradigm.

The CIP project was generally successful, and its investigators are optimistic about the general approach. We intend to study their system carefully in the development of Breitenbush and Santiam.

Our use of the Cornell Synthesizer Generator is largely inspired by the experience of the Penelope group at Odyssey Research Associates[1][14]. During Penelope's early development its implementors were particularly pleased with the synthesizer; the use of the right tool "jump-started" the project. The Penelope system provides an environment for developing Ada programs that are annotated with assertions. It is not a true inferential system since it does not maintain the development from specification to code.

There are several interactive program transformation systems in which executable specifications are automatically or semi-automatically improved, including the KIDS system developed by Kestrel Institute[48]. We plan to develop tools for program transformation as part of the larger OGI effort on formal methods, but we do not wish to restrict ourselves to executable specification languages.

Programming logics with similar goals to EML include Z, VDM and Larch. Z[49, 50] and VDM[18] are the most distinct from EML. They are language independent notations derived from set theory. They have been used quite successfully in rigorous environments, where formal reasoning is done in the specification logic, generally without automatic support. In these examples programs are more informally associated with their specifications. The logics are typically not integrated with a specific programming language. There has been some work on tools for Z and VDM, including the ESPRIT project RAISE. RAISE has produced an interesting, but rather complicated, specification language and methodology[28]. Any serious research into practical specification environments must study the successful aspects of Z and VDM carefully. However, we do not feel they are the ultimate answer.

Larch is a hybrid system[15, 51]. It consists of a language independent "shared language" and a collection of language dependent "interface languages." Top level specifications are given in the shared language. As programs are developed, specifications in the shared language are refined into language specific interface specifications. By attempting to interface to many languages, most of which were not defined with verification in mind, Larch is trying to solve a fundamentally harder problem than we propose. As with the other systems mentioned, Larch has had important successes and merits study during this investigation.

The most closely related work on EML is being undertaken by Sannella's group at

---

1My former employer.
Edinburgh. They are currently focused on the semantic definition of EML for larger subsets of SML, modifying compiler tools to support the type checking and execution of partially-executable EML specifications, and interfacing these compiler tools to theorem-proving environments. The areas of emphasis identified in this proposal are meant to complement the efforts in Edinburgh.

7 Research plan

The first task is the implementation of Breitenbush. A masters student has already initiated construction of a synthesizer generated editor for Standard ML. If that project is successful it will become the core of the Breitenbush implementation. I expect Breitenbush to be operational within the first three to six months of the project. Once Breitenbush is available a series of examples will be worked out and the EML methodology will be exercised. We expect to discover idioms of use and identify useful support tools that we can incorporate into Breitenbush. At the end of the first year we will stop development of Breitenbush.

In parallel with the implementation, we will begin investigating the theoretical questions outlined in Section 5. At the end of the first year we will have evaluated the multi-spectral approach and be ready to specify the logical support required for Santiam.

When these goals have been met we plan to visit Sannella's group in Edinburgh to evaluate our progress and discuss strategy. At that point we will have written reports documenting Breitenbush and presenting a collection of case studies.

The second year of the project will focus on the design and partial implementation of Santiam. The first question to be addressed will be how to maintain the methodological benefits obtained from the attribute grammar implementation of Breitenbush.

Other issues to be investigated in the design of Santiam will include the choice of a formal logic support environment, the architecture of multi-spectral logic support, Santiam's relationship to the Edinburgh tools, and the potential of integrating the system with the larger OGI effort on Formal Methods.

By the middle of the second year we will have formalized the design of some Santiam components and begun their development in Breitenbush. At the end of that year we will have a complete design for Santiam and an implementation of some system components.

Theoretical investigations will continue in the second year as required to support the design of Santiam and the refinement of the methodology.

Throughout the investigation we will report progress in conference proceedings and journals and market this research to industry and other government agencies. If additional funding is obtained, and the level of effort can be increased, we will accelerate the schedule accordingly.
A Appendix: A simple example

This appendix illustrates a few features of Standard ML and presents the derivation of a quick sort algorithm in Extended ML. It briefly describes the Breitenbush tool and its utility in the derivation process. It may be read independently of the body of the report.

A.1 Standard ML

ML is a polymorphic, call-by-value functional programming language with references and exceptions. Although the language is strongly-typed, ML programs are expressed with minimal type annotations. Type inference is performed by the compiler using a complete algorithm based on unification[22]. Functions in ML may either be defined by lambda abstraction or by cases on the structure of the arguments. For example, the identity function, which simply returns its argument, is written in the lambda-calculus \( \lambda x.x \) and in Standard ML as

```ml
fun length [] = 0
| length (x::xs) = 1 + length xs
```

Here `::` is the infix list “cons” operator and `[]` represents “nil.” Note that the variables occurring in the pattern `(x::xs)` on the left hand side of the equal sign are bound on the right. The polymorphic types inferred for the two examples are \( \text{'}a \rightarrow \text{'}a \) for the identity and `length : 'a list -> int` for the length function. Type variables, indicated with a leading “\( \text{'} \)”, range over all (mono) types in the language. Note that no type annotations are present in the two programs. Type annotations are only needed in SML to resolve overloaded identifiers.

The module facility of SML has as its basic entity the structure. A structure may be viewed as a “large value” that collects together types and small values, as well as other structures from the module language. The visible contents of a structure are described by a signature. Signatures are “large types;” they collect the names and types of components of structures. This method of data encapsulation is similar to that used in object oriented programming, however structures are less dynamic in nature and have a significantly simpler semantic interpretation. Structures are not first-class runtime objects (i.e. they cannot be passed as values to functions at run-time); they are, however, first-class objects at compile time (or “link time”).

The third and final entity in the module facility is the functor. Functors map structures to structures; they may be viewed as “large functions.” Functors are defined as abstracted structures or as compositions of functors. All functor applications are elaborated at compile time. Functors are similar to generic encapsulation features in Ada. However, they are both more uniform and more general than the mechanisms provided by Ada.
In the final report on SML it is clear that Milner achieved the goal of developing a formally defined, elegant language that could be used by real programmers[23]. There are currently at least three major implementations of SML—the Edinburgh implementation, MacQueen and Appel’s Standard ML of New Jersey[1], and Matthews’ POLY implementation. In addition there is at least one private company, Harlequin Ltd of Cambridge, England, developing a commercial implementation. In the eyes of many, SML is the language design success story of the 1980s.

A.2 Extended ML

Extended ML (EML) weds the many-sorted algebraic specification technology of Goguen, Burstall, Sannella, Tarlecki, and Ehrig with SML[6, 38]. The name Extended ML applies to a family of specification languages, satisfaction relations and associated program development methodologies being investigated by Sannella and Tarlecki[41, 42, 39, 44]. All of the languages in the EML family are obtained by picking a logic expressive over a subset of the ML core language and extending it with the SML module facility. The methodologies provide the same three simple steps: decomposition, coding and refinement. Each step insures that the derived program satisfies the specification, provided the associated proof obligations are discharged. This report focuses on tools to automate and record the derivation process.

A.2.1 The EML specification language

The EML language is obtained from SML by extending the module facility with axioms describing program behavior. As expected, this extends SML signatures to traditional algebraic specifications and functors to maps between structures satisfying specifications. For example, to specify sorting, we would derive a functor from a total order to a total order with a sort operation. The total order would be specified:

```ml
signature Total_Order =
  sig type elem
    val <= : elem * elem -> bool
    axiom all x. x <= x
    axiom all x,y. x <= y andalso y <= x implies x = y
    axiom all x,y,z. x <= y andalso y <= z implies x <= z
    axiom all x,y. x <= y orelse y <= x
  end

The sorted total order:

signature Sorted_Total_Order =
```
sig include Total_Order
  val sort : elem list -> elem list
axiom all l. permutation 1 (sort l)
  andalso ordered (sort l)
end

We are assuming that ordered and permutation have been previously defined. The include directive repeats the specification of total orders as part of the specification of sorted total orders. With these definitions in place, the functor sort can be specified:

functor sort (0:Total_Order) :
  sig include Sorted_Total_Order
    sharing type 0.elem = elem
  end
= ?

The placeholder, ?, is an important part of the EML language. It indicates that it is undetermined how to implement the sorting functor. Also note the sharing declaration in the result type of the functor. This requires that the result of the sort functor be related to its argument.

A novel feature of EML is that axioms are also allowed in structures, where they can be used to describe classes of implementations. Returning to our example, EML will allow the following "implementation" of the sort functor:

functor sort (0:Total_Order) :
  sig include Sorted_Total_Order
    sharing type 0.elem = elem
  end
= struct
    type elem = 0.elem
    val sort : elem list -> elem list = ?
    axiom all l. permutation 1 (sort l)
      andalso ordered (sort l)
  end

This "implementation" of sort is constrained only by the axioms. The meaning given such underspecified structures is the class of all SML structures that satisfy the constraints (up to behavioral equivalence\(^2\)). These underspecified implementations are exploited in the refinement methodology, where users successively introduce constraints until they have a completely specified SML structure that satisfies its specification.

\(^2\)This is suggestive, but incomplete. See the foundations paper for details[44].
A.2.2 The EML methodology

The EML methodology is "top down." It starts with a specification of the entire program, decomposes this into successively smaller subproblems, and then refines the subproblems into code. Program derivations in the methodology may be viewed as a tree of functor refinements of three basic sorts: decomposition steps, coding steps and refinement steps. The decomposition steps involve the module language exclusively. They implement a functor by composing two simpler functors, typically introducing a new signature for the interface of the two newly specified functors. The coding steps are the transition from the module language level to the core. They provide an abstract functor body to implement a functor. The refinement steps take abstract functor bodies to more concrete ones, eventually producing SML code.

For example, if we decide to derive the quick sort algorithm as our implementation of the sort functor, we would first decompose the problem into producing partitioned total orders and then exploit the partitioned orders in the implementation of sorting. This decomposition step requires the interface signature:
signature Partitioned_Total_Order =
  sig include Total_Order
  val partition : elem -> elem list -> (elem list * elem list)
  axiom all a,l.
    let val at_or_below,above = partition a l
    in permutation l (at_or_below@above)
    andalso
      all x. (x is_in at_or_below implies x <= a)
    andalso (x is_in above implies not (x <= a))
  end
end

The sort functor is now implemented by two new functors, partition and qsort as follows:

functor partition (O:Total_Order) :
  sig include Partitioned_Total_Order
  sharing type O.elem = elem
  end
  = ?

functor qsort (O:Partitioned_Total_Order) :
  sig include Sorted_Total_Order
  sharing type O.elem = elem
  end
  = ?

functor sort (O:Total_Order) :
  sig include Sorted_Total_Order
  sharing type O.elem = elem
  end
  = qsort(partition(O))
Coding steps replace a placeholder (?) functor body by a structure. For example, we can code the qsort functor

```sml
functor qsort (0:Partitioned_Total_Order) :
  sig include Sorted_Total_Order
    sharing type 0.elem = elem
  end
  = struct
    open 0
    val sort : elem list -> elem list = ?
    axiom all 1. permutation 1 (sort 1)
      andalso ordered (sort 1)
  end
```

The `open` declaration includes all components of 0 in the structure being created. The partition operation of 0 will not be exported because it is not mentioned in the Sorted_Total_Order signature.

This implementation can be made more specific by a refinement step:

```sml
functor qsort (0:Partitioned_Total_Order) :
  sig include Sorted_Total_Order
    sharing type 0.elem = elem
  end
  = struct
    open 0
    fun sort [] = []
    | sort [a] = [a]
    | sort (a::l) = let val (at_or_below, above) = partition a 1
                   in (sort at_or_below)@[a]@[sort above)
                   end
  end
```

This final step produces executable SML code. Each development step gives rise to a number of proof obligations which can be generated automatically from the "before" and "after" versions of the functor. These proof obligations are trivially satisfied except in the last step. The proof obligation in the last step is to show that the new body of qsort, plus the axioms in Partitioned_Total_Order, entail the old body of qsort; this follows by induction. To complete the example the reader is invited to derive the implementation of the partition functor.

It is not uncommon in the design process to discover that an interface specification needs to be relaxed or strengthened, or that an alternative decomposition might
simplify the derivation. Such modifications may cause a cascade of changes within the derivation. It is illuminating to consider how the example would be modified to specify stable sorting, that is, sorting where the relative order of equivalent elements remains unchanged. However, as the reader may have already discovered, stable sorts do not sort total orders! The antisymmetry law, \( \forall x, y. \ x \leq y \text{ and also } y \leq x \implies x = y \), is too strong to allow stable sort to be defined. Instead one must start with a total quasi-order (a total, reflexive, transitive relation). As a result, some logical inferences may no longer be established by identity and verification conditions must be updated.

While the EML methodology of program development is top down and based on decomposition, it can be viewed as a system that promotes the composition of reusable program components. An expert working in an environment with a rich library of specifications and functors can guide the decomposition into previously defined concepts. Since the library records the specifications of modules, as well as the signatures and implementations, code can safely be reused and code with multiple uses may be safely maintained and improved (provided the specification is not violated).

### A.3 Breitenbush — a derivation editor

Our initial tool, the Breitenbush derivation editor, will be built with the Cornell synthesizer generator\cite{35, 34, 36, 37}. The synthesizer generator is a tool that produces interactive structured editors from language specifications given in an attribute grammar-based formalism. We feel, based on the experience of Griffin\cite{12} and of Guaspari, Marceau and Pollak\cite{14}, that this will be an appropriate environment for rapidly prototyping an editor interface for EML.

Breitenbush will support the derivation process by maintaining the tree of derivation steps interactively. In the sort example, the derivation begins with the \texttt{sort} specification. The editor will allow us to indicate a decomposition step, splitting the derivation into the \texttt{partition} and \texttt{qsort} subderivations. When we indicate the trivial coding step for the \texttt{qsort} functor, it will automatically generate the subderivation starting with the unconstrained implementation. When this implementation is refined into code, the editor will compute the verification conditions introduced by the refinements. (We do not expect the editor to do any non-trivial theorem proving.) Another attribute computed by the editor will be the executable SML code corresponding to the derivation.

The real benefit of the synthesizer technology, however, comes when we want to propagate changes through the derivation. For the stable sort modification, the synthesizer's attribute propagation facility will allow us to make this specification change at the top level and have it propagate throughout the derivation. Breitenbush will automatically highlight logical inferences that are no longer established by identity and update the list of verification conditions.

This mechanism will also be helpful as designers explore the design space and the
impact of design decisions. Program specification and synthesis does not eliminate error in specification and design—it simply promotes the discovery of such errors by making claims about the abstract behavior of systems explicit and providing tools for the analysis of the consistency of these claims. It is important to note that these discoveries are made by reasoning at design time, not by testing executing code. This is as radical a departure from the test and debug technology used today as design rule checking is from testing fabricated chips.

It is not uncommon in the design process to discover that an interface specification needs to be relaxed or strengthened, or that an alternative decomposition might simplify the derivation. When doing derivations by hand, such changes are difficult because it is tedious to predict their ramifications. This is another situation in which the attribute computation engine provided by a synthesizer-generated editor is expected to have a large payoff.

There is also a potential for design and derivation reuse. In addition to the specification and implementations of components, the library will also contain the complete derivations of modules and systems. This will often include important intermediate abstractions that are not apparent in either the specification or the final implementation. The stable sorting exercise is an example of design reuse.

Tools for support of EML are in their infancy. Currently Kazmierczak, under the direction of Sannella, is working on a parser and interface to various theorem proving engines, including LEGO and Isabelle[40, 32, 21, 30]. In spite of the lack of tools, EML is being used in program development by Harlequin Ltd of Cambridge[47].

References


