Fully Automatic Adaptation of Software Components Based on Semantic Specifications *

Christian Haack[2], Brian Howard[1], Allen Stoughton[1], and J. B. Wells[2]

[1] Kansas State University
http://www.cis.ksu.edu/santos/
[2] Heriot-Watt University
http://www.cee.hw.ac.uk/ultra/

Abstract. We describe the design and methods of a tool that, based on behavioral specifications in interfaces, generates simple adaptation code to overcome incompatibilities between Standard ML modules.

1 Introduction

1.1 The Problem of Unstable Interfaces

The functionality of current software systems crucially depends on the stability of module interfaces. Whereas implementations of interfaces may change, currently deployed software technology requires that the interfaces themselves remain stable because changing an interface without changing all of the clients (possibly around the entire world) results in failure to compile. However, in practice it is often the case that interfaces gradually change and improve. It may, for example, turn out that it is useful to add a few extra parameters to a function in order to make it more generally applicable. Or, it may be more convenient to give a function a different, but isomorphic, type than initially specified. In this paper, we address the issue of unstable interfaces in the context of Standard ML (SML) and propose a language extension that allows changing module interfaces in certain simple but common ways without needing the module clients to be modified.

As a simple example, consider the interface of a module that contains a sorting function for integer lists. Using SML syntax, such an interface looks like this:[1]

```
signature SORT = sig   val sort : int list -> int list   end
```

Suppose that in a later version this sorting function is replaced by a more general function that works on lists of arbitrary type and takes the element ordering as a parameter:

* This work was partially supported by NSF/DARPA grant CCR-9633388, EPSRC grant GR/R 41545/01, EC FP5 grant IST-2001-33477, NATO grant CRG 971607, NSF grants CCR 9988529 and ITR 0113193, Sun Microsystems equipment grant EDUD-7826-990410-US.

[1] Modules are called *structures* in SML and interfaces (the types of modules) are called *signatures*.

```
signature SORT = sig
  val sort : ('a * 'a -> bool) -> 'a list -> 'a list
end
```

In the example, `'a` is a type variable. `sort` is now a polymorphic function, and has all types that can be obtained by instantiating `'a`. Implementations of the more general of these two interfaces can easily be transformed into implementations of the more special one: Just apply the general sorting function to the standard ordering on the integers. However, the SML compiler does not recognize this. Programs that assume the special interface will fail to compile when the interface is replaced by the general one.

One way to ensure compatibility with the rest of the system when an interface is changed is to always keep the existing functions. For example, when a function (and its type specification) is generalized, the older and more specialized version can be kept. However, this method results in redundancy in the interfaces, which is hardly desirable. Therefore, with currently deployed software technology, it is probably better to change the interfaces, rely on the compiler to report the places where the old interface is assumed, and then fix those places by hand to work with the new interface. Such a manual adaptation is not hard for someone familiar with the programming language, but requires a human programmer and is not automated. It is this step of bridging gaps between interfaces that our tool automates. It adapts a software component by synthesizing wrapper code that transforms implementations of its interface into implementations of a different, but closely related, interface.

1.2 The AxML Language for Synthesis Requests

Extending Interfaces by Semantic Specifications. In the design of our tool, we were guided by the following two principles: Firstly, component adaptation should solely be based on information that is explicitly recorded in interfaces. Secondly, the synthesized interface transformations should transform semantically correct implementations into semantically correct implementations. The emphasis in the second principle is on *semantically correct* as opposed to *type correct*. Considering these two principles, it becomes clear that we have to add semantic specifications into interfaces.[2] For this purpose, we have designed the language AxML ("**A**nother e**x**tended **ML**"), an extension of SML with specification axioms.

Specification axioms are certain formulas from predicate logic. In order to convey an idea of how they look, we extend the interface from above by a specification axiom. As basic vocabulary, we use the following atomic predicates:

```
linOrder : ('a * 'a -> bool) -> prop
sorted : ('a * 'a -> bool) -> 'a list -> prop
bagEq : 'a list -> 'a list -> prop
```

[2] One could view these semantic specifications as part of the type of a module. However, in this paper we use the following terminology: We refer to the traditional SML types as types, whereas we refer to the AxML specification axioms as semantic specifications.

The type `prop` represents the two-element set of truth values, and predicates denote truth-valued functions. The intended meanings of the above atomic predicate symbols are summarized in the following table:

`linOrder f`	`f` is a linear order
`sorted f xs`	`xs` is sorted with respect to `f`
`bagEq xs ys`	`xs` and `ys` are equal as multisets

The AxML system has been deliberately designed to never need to know the meaning of these predicate symbols. Instead, component adaptation works correctly for any valid assignment of meanings to predicate symbols.

Using this vocabulary, we now extend the general `SORT` interface by an axiom. `fa` denotes universal quantification, `->` implication and `&` conjunction.

```
signature SORT = sig
  val sort : ('a * 'a -> bool) -> 'a list -> 'a list
  axiom  fa f : 'a * 'a -> bool, xs : 'a list =>
           linOrder f -> sorted f (sort f xs) & bagEq xs (sort f xs)
end
```

We say that a module *implements* an interface if it contains all the specified functions with the specified types (*type correctness*), and, in addition, the functions satisfy all the specification axioms that are contained in the interface (*semantic correctness*). AxML verifies type correctness of a module, but does not verify semantic correctness. It is the programmer's responsibility to ensure the semantic correctness of implementations. It is important to understand that although the traditional SML type information appearing in the interface of a module will be verified, our system *deliberately* does not attempt to verify that a module satisfies the semantic specification axioms that appear in its interface. Our adaptation tool preserves semantic correctness *if it already holds*. This differs from recent work in Nuprl [3] where the implementation of an interface must contain a proof for each axiom.

AxML provides mechanisms to introduce atomic predicate symbols, like, for example, `linOrder`, `sorted` and `bagEq`. The extensive use of such predicate symbols in specification formulas, reflects, we think, the natural granularity in which programmers would write informal program documentation: When describing the behavior of a sorting function, a programmer would use the word "sorted" as basic vocabulary, without further explanation. (The detailed explanation of this word may have been given elsewhere, in order to ensure that programmers agree on its meaning.) We hope that a good use of AxML results in specification axioms that closely resemble informal documentation. From this angle, AxML can be viewed as a formal language for program documentation. Writing documentation in a formal language enforces a certain discipline on the documentation. Moreover, formal documentation can be type-checked, and some flaws in the documentation itself can be discovered that way.

Synthesis Requests. It is now time to describe AxML's user interface: The user will supply AxML files. Every SML file is an AxML file, but, in addition, AxML files can contain semantic specifications in interfaces. Moreover,

AxML files can contain so-called *synthesis requests* that request the adaptation of a set of modules to implement a certain interface. An AxML "compiler" will translate AxML files to SML files by simply dropping all semantic specifications and replacing synthesis requests by implementations that have been generated by a synthesis tool. The synthesized code is of a very simple kind. Typically, it will specialize by supplying arguments, curry, uncurry or compose existing functions. Synthesized code is not recursive. Its purpose is to transform modules into similar modules.

Here is an example of a synthesis request. Assume first that `Sort` is a module that implements the second version of the `SORT` interface. Assume also that `Int` is a module that contains a function `leq` of type `int * int -> bool`, and that `Int`'s interface reports that this function is a linear order. Here is the AxML interface `MYSORT` for an integer sort function:

```
signature MYSORT = sig
  val sort : int list -> int list
  axiom fa xs : int list => sorted Int.leq (sort xs) & bagEq xs (sort xs)
end
```

The following AxML declaration contains a synthesis request for an implementation of `MYSORT`. The synthesis request is the statement that is enclosed by curly braces.

```
structure MySort : MYSORT = { fromAxioms   structure Sort  structure Int
                              create  MYSORT }
```

The AxML compiler will replace the synthesis request by an implementation of the interface `MYSORT`:

```
 structure MySort : MYSORT = struct   val sort = Sort.sort Int.leq   end
```

Here is how AxML can be used to protect a software system against certain interface changes: If a module's interface is still unstable and subject to change, the programmer may use it indirectly by requesting synthesis of the functions that they need from the existing module. Initially, it may even be the case that these functions are exactly contained in the module. (In this case, the "synthesized" functions are merely selected from the existing module.) If the interface of the existing module changes in a way that permits the recovery of the old module by simple transformations, the functions will automatically be resynthesized. In this manner, the AxML compiler overcomes what would be a type incompatibility in currently deployed systems.

The *simple transformations* that AxML will be able to generate consist of expressions that are built from variables by means of function abstraction, function application, record formation and record selection. AxML will deliberately not attempt to generate more complex transformations, because that seems too hard to achieve fully automatically.

```
signature BINREL = sig
  type 'a t = 'a * 'a -> bool
  pred 'a linOrder : 'a t -> prop
  pred 'a linPreorder : 'a t -> prop
  pred 'a preorder : 'a t -> prop
  pred 'a equivalence : 'a t -> prop
  val symmetrize : 'a t -> 'a t
  axiom  fa r : 'a t => linOrder r -> linPreorder r
  axiom  fa r : 'a t => linPreorder r -> preorder r
  axiom  fa r : 'a t => preorder r -> equivalence (symmetrize r)
end

structure BinRel : BINREL = struct
  type 'a t = 'a * 'a -> bool
  fun symmetrize f (x,y) = f(x,y) andalso f(y,x)
end
```

Fig. 1. A structure of binary relations

Code synthesis of this kind is incomplete. Our synthesis algorithm can be viewed as a well-directed search procedure that interrupts its search after a certain time, possibly missing solutions that way.

A Further Benefit: Recognition of Semantic Errors. Because component adaptation is based on behavioral specifications rather than just traditional types, AxML will have another desirable effect: Sometimes a re-implementation of a function changes its behavior but not its traditional type. When this happens, other parts of the system may behave incorrectly because they depended on the old semantics. In traditional languages, such errors will not be reported by the compiler. AxML, on the other hand, fails to synthesize code unless correctness is guaranteed. Therefore, if behavioral changes due to re-implementations are recorded in the interfaces, then AxML will fail to resynthesize requested code if the changes result in incorrect program behavior. In this manner, the AxML compiler reports incompatibilities of behavioral specifications.

2 An Overview of AxML

User-Introduced Predicate Symbols. AxML provides the programmer with the facility to introduce new predicate symbols for use in specification axioms. For example, the structure in Figure 1 introduces predicate symbols and functions concerning binary relations. Predicates may have type parameters. Thus, a predicate symbol does not denote just a single truth-valued function but an entire type-indexed family. For example, `linPreorder` has one type parameter `'a`. In predicate specifications, the list of type parameters must be mentioned explicitly. The general form for *predicate specifications* is

$$\texttt{pred (}\, a_1, \ldots, a_n \,\texttt{)}\ id : predty$$

where $a_1, \ldots, a_n$ is a sequence of type variables, *id* is an identifier and *predty* is a predicate type, i.e., a type whose final result type is `prop`. The parentheses that enclose the type variable sequence may be omitted if the sequence consists of a single type variable and must be omitted if the sequence is empty.

When a structure S is constrained by a signature that contains a specification of a predicate symbol *id*, a new predicate symbol is introduced into the environment and can be referred to by the long identifier *S.id*. For instance, the structure declaration in the example introduces the predicate symbols `BinRel.linOrder`, `BinRel.linPreorder`, `BinRel.preorder` and `BinRel.equivalence`.

AxML permits the explicit initialization of type parameters for predicate symbols. The syntax for initializing the type parameters of a predicate symbol *longid* by types $ty_1, \ldots, ty_n$ is

$$longid \texttt{ at (} ty_1, \ldots, ty_n \texttt{)}$$

The parentheses that enclose the type sequence may be omitted if the sequence consist of a single type. For example, (`BinRel.linPreorder at int`) initializes the type parameter of `BinRel.linPreorder` by the type `int`.

From AxML's point of view, predicate specifications introduce uninterpreted atomic predicate symbols. AxML synthesis is correct for *any* valid assignment of meanings to predicate symbols. On the other hand, programmers that introduce such symbols should have a concrete interpretation in mind. It is, of course, important that different programmers agree on the interpretation. The interpretations of the predicate symbols from the example are as follows:
(`(BinRel.linPreorder at` *ty*`)` r) holds if and only if the relation r is a linear preorder at type *ty*, i.e., it is reflexive, transitive and for every pair of values (x, y) of type *ty* it is the case that either $r(x, y) =$ `true` or $r(y, x) =$ `true`. Similarly, the interpretation of `BinRel.linOrder` is "is a linear order", the interpretation of `BinRel.preorder` is "is a preorder", and the interpretation of `BinRel.equivalence` is "is an equivalence relation".

Formally, specification predicates are interpreted by sets of closed SML expressions that are closed under observational equality. For instance, it is not allowed to introduce a predicate of type `(int list -> int list) -> prop` that distinguishes between different sorting algorithms. All sorting algorithms are observationally equal, and, therefore, a predicate that is true for merge-sort but false for insertion-sort, call it `isInNlogN`, is not a legal specification predicate. The reader who is interested in a detailed treatment of the semantics of specification formulas is referred to [7].

There is a built-in predicate symbol for observational equality. It has the following type specification:

```
pred 'a == : 'a -> 'a -> prop
```

In contrast to user-introduced predicate symbols, `==` is an infix operator. The AxML synthesizer currently treats `==` like any other predicate symbol and does not take advantage of particular properties of observational equality.

Specification Axioms. Predicate symbols are turned into *atomic specification formulas* by first initializing their type parameters and then applying them to a sequence of terms. Here, the set of *terms* is a small subset of the set of SML expressions. It consists of all those expressions that can be built from variables and constants, using only function application, function abstraction, record formation and record selection. The set of terms coincides with the set of those

SML expressions that may occur in synthesized transformations. *Specification formulas* are built from atomic specification formulas, using the propositional connectives and universal quantification. For example, if `Int.even` is a predicate symbol of type `int -> prop`, then the following is a well-typed specification formula:

```
fa x : int => Int.even ((fn y => y + y) x)
```

On the other hand, the following is *not* a specification formula, because the argument of `Int.even` is not a term:

```
fa x : int => Int.even (let fun f x = f x in f 0 end)
```

A universally quantified formula is of one of the following three forms:

`fa` *id* `:` *ty* `=>` F `fa` *id* `:` $([a_1, \ldots, a_n]\ .\ ty)$ `=>` F `fa` a `=>` F

where *id* is a value identifier, *ty* is a type, $a, a_1, \ldots, a_n$ is a sequence of type or equality type variables and F is a formula. The first two forms denote *value quantification* and the third one *type quantification*. The expression $([a_1, \ldots, a_n]\ .\ ty)$ denotes an SML type scheme, namely, the type scheme of all values that are of type *ty* and polymorphic in the type variables $a_1, \ldots, a_n$. Quantification over polymorphic values is useful, because it allows to express certain theorems — so-called "free" theorems — that hold exactly because of the polymorphism [22]. Value quantification ranges over all *values* of the specified type or type scheme, as opposed to all, possibly non-terminating, closed expressions.

Type parameters of atomic predicate symbols may be explicitly instantiated, but they do not have to be. A formula where type instantiations are left implicit is regarded as a shorthand for the formula that results from inserting the most general type arguments that make the formula well-typed. For example,

```
fa f : 'a * 'a -> bool, xs : 'a list =>
  BinRel.linOrder f ->
    List.sorted f (sort f xs) & List.bagEq xs (sort f xs)
```

is a shorthand for

```
fa f : 'a * 'a -> bool, xs : 'a list =>
  (BinRel.linOrder at 'a) f ->
    (List.sorted at 'a) f (sort f xs) & (List.bagEq at 'a) xs (sort f xs)
```

Free type variables in formulas are implicitly all-quantified at the beginning of the formula. Therefore, the previous formula expands to

```
fa 'a => fa f : 'a * 'a -> bool, xs : 'a list =>
  (BinRel.linOrder at 'a) f ->
    (List.sorted at 'a) f (sort f xs) & (List.bagEq at 'a) xs (sort f xs)
```

```
signature TABLE = sig
  type (''k,'v) t
  val empty : (''k,'v) t
  val update : (''k,'v) t -> ''k -> 'v -> (''k,'v) t
  val lookup : (''k,'v) t -> ''k -> 'v option
  axiom  fa x : ''k => lookup empty x == NONE
  axiom  fa x : ''k, t : (''k,'v) t, y : 'v =>
           lookup (update t x y) x == SOME y
  axiom  fa x1,x2 : ''k, t : (''k,'v) t, y : 'v =>
           Not (x1 == x2) -> lookup (update t x1 y) x2 == lookup t x2
end
```

Fig. 2. An interface of lookup tables

Example 1. The signature in Figure 2 specifies an abstract type `t` of lookup tables. The type `t` has two type parameters, namely `''k`, the type of keys, and `'v`, the type of values. Type variables that begin with a double prime range over equality types only. Equality types are types that permit equality tests. For example, the type of integers and products of equality types are equality types, but function types are not. Elements of the datatype `'b option` are either of the form `SOME` x, where x is an element of type `'b`, or they are equal to `NONE`. The keyword `Not` denotes propositional negation.

Synthesis Requests. Synthesis requests are always surrounded by curly braces. They come in three flavors: As requests for structure expressions, functor bindings and structure level declarations. Thus, we have extended the syntax classes of structure expressions, functor bindings and structure-level declarations from the SML grammar [15] by the following three phrases:

$$\begin{array}{lll} \mathit{strexp} & ::= \ldots & \mid \texttt{\{}\ \mathit{createstrexp}\ \texttt{\}} \\ \mathit{funbind} & ::= \ldots & \mid \texttt{\{}\ \mathit{createfunbind}\ \texttt{\}} \\ \mathit{strdec} & ::= \ldots & \mid \texttt{\{}\ \mathit{createstrdec}\ \texttt{\}} \end{array}$$

The synthesis requests are of the following forms:

$$\begin{array}{ll} \mathit{createstrexp} & ::= \langle \texttt{fromAxioms}\ \mathit{axiomset} \rangle\ \texttt{create}\ \mathit{sigexp} \\ \mathit{createfunbind} & ::= \langle \texttt{fromAxioms}\ \mathit{axiomset} \rangle\ \texttt{create}\ \mathit{funspec} \\ \mathit{createstrdec} & ::= \langle \texttt{fromAxioms}\ \mathit{axiomset} \rangle\ \texttt{create}\ \mathit{spec} \end{array}$$

The parts between the angle brackets $\langle\ \rangle$ are optional. The syntax domains of signature expressions *sigexp* and specifications *spec* are the ones from SML, but enriched with axioms and predicate specifications. The domain *funspec* of functor specifications is not present in SML. Functor specifications are of the form

$$\mathit{funspec} ::= \mathit{funid}\ \texttt{(}\ \mathit{spec}\ \texttt{)}\ \texttt{:}\ \mathit{sigexp}$$

where *funid* is a functor identifier. *Axiom sets* are lists of specification formulas, preceded by the keyword `axiom`, and structure identifiers, preceded by the keyword `structure`. A structure identifier introduces into the axiom set all the specification formulas that are contained in the structure that the identifier

```
signature TABLE' = sig
  type ('v,''k) t
  val empty : ('v,''k) t
  val update : ('v,''k) t -> ''k * 'v -> ('v,''k) t
  val lookup : ''k -> ('v,''k) t -> 'v option
  axiom  fa x : ''k => lookup x empty == NONE
  axiom  fa x : ''k, t : ('v,''k) t, y : 'v =>
           lookup x (update t (x,y)) == SOME y
  axiom  fa x1,x2 : ''k, t : ('v,''k) t, y : 'v =>
           Not (x1 == x2) -> lookup x2 (update t (x1,y)) == lookup x2 t
end
```

Fig. 3. Another interface of lookup tables

refers to. (An axiom is contained in a structure if it is contained in its explicit signature.)

When encountering a synthesis request, the AxML compiler creates a structure expression (respectively functor binding, structure level declaration) that implements the given signature (respectively functor specification, specification). The created implementation of the signature is guaranteed to be semantically correct, *provided* that the environment correctly implements the axioms in *axiomset*. A synthesis may fail for several reasons: Firstly, there may not exist a structure expression that satisfies the given specification. Secondly, a structure expression that satisfies the given specification may exist, but its correctness may not be provable from the assumptions in *axiomset*. Thirdly, a structure expression that satisfies the given specification may exist and its correctness may be provable from the given axiom set, but the synthesizer may not find it due to its incompleteness.

Example 2. Figure 3 shows an interface of lookup tables that differs from the one in Figure 2. The functions in the two signatures differ by type isomorphisms. Moreover, the type parameters of the type `t` appear in different orders. Here is a request for creation of a functor that transforms `TABLE'` structures into `TABLE` structures:

```
functor { create F ( structure T : TABLE' ) : TABLE }
```

Here is the synthesized functor:

```
functor F ( structure T : TABLE' ) : TABLE = struct
  type (''k,'v) t = ('v,''k) T.t
  val empty = T.empty
  val update = fn t => fn x => fn y => T.update t (x,y)
  val lookup = fn t => fn x => T.lookup x t
end
```

Note that this example demonstrates that AxML is capable of synthesizing type definitions.

3 Synthesis Methods

This section gives an overview of the methods. Details can be found in [7]. Our methods and the style of their presentation has been inspired by the description of a λProlog interpreter in [14].

3.1 Sequent Problems

Synthesis requests get translated to sequent problems. A *sequent problem* is a triple of the form

$$\mathcal{C}; (\Delta \vdash \Theta)$$

where $\mathcal{C}$ is a problem context, and both Δ and Θ are finite sets of formulas. The *problem context* $\mathcal{C}$ is a list of declarations. A *declaration* assigns types schemes to all value variables, and kinds to all type function variables that occur freely in Δ or Θ. *Kinds* specify the arity of type constructors. Typical kinds are the kind **type** of types, the kind **eqtype** of equality types and type function kinds like **type** → **type** or **eqtype** → **type** → **eqtype**. For example, the kind of type constructor `t` in Figure 2 is **eqtype** → **type** → **type** (which happens to have the same inhabitants as **type** → **type** → **type**).

A declaration also tags each variable with either a ∀-symbol or an ∃-symbol. These are just tags and are not to be confused with quantifiers in specification formulas. Variables that are tagged by an ∃ are called *existential variables* and variables that are tagged by a ∀ are called *universal variables*. Universal variables may be viewed as fixed parameters or constants. Existential variables, on the other hand, are auxiliary variables that ought to be substituted. It is the goal of our methods to replace the existential variables by terms that only contain universal variables. These substitution terms constitute the implementations of the existential variables. The universal variables correspond to preexisting resources that can be used in the synthesis and the existential variables correspond to the values the user has requested to be synthesized. The order of declarations in the variable context $\mathcal{C}$ is important because it encodes scoping constraints: Substitution terms for an existential variable x may only contain universal variables that occur before x in $\mathcal{C}$.

Example 3. The sequent problem that results from translating the synthesis request from Example 2 has the shape $(\mathcal{C}; (\Delta \vdash \{F\}))$, where Δ is the set of all specification axioms that are contained in signature `TABLE'`, F is the conjunction of all specification axioms that are contained in signature `TABLE`, and $\mathcal{C}$ is the following problem context:

∀`T.t` : **type** → **eqtype** → **type**,
∀`T.empty` : `('v,''k) T.t`,
∀`T.update` : `('v,''k) T.t -> ''k * 'v -> ('v,''k) T.t`,
∀`T.lookup` : `''k -> ('v,''k) T.t -> 'v option`,
∃`t` : **eqtype** → **type** → **type**,
∃`empty` : `(''k,'v) t`,
∃`update` : `(''k,'v) t -> ''k -> 'v -> (''k,'v) t`,
∃`lookup` : `(''k,'v) t -> ''k -> 'v option`

A *sequent* is a sequent problem whose problem context does not declare any existential variables. A sequent $(\mathcal{C}; (\Delta \vdash \Theta))$ is called *valid* iff for all interpretations of its universal variables by SML values, all interpretations of its universal type function variables by SML type definitions and all interpretations of atomic predicate symbols by specification properties, the following statement holds:

If *all* formulas in Δ are true, then *some* formula in Θ is true.

A solution to a sequent problem $\mathcal{P}$ is a substitution s for the existential variables, such that the sequent that results from applying s to $\mathcal{P}$ and removing the existential variables from $\mathcal{P}$'s problem context is valid. Here is the solution to the sequent problem from Example 3. The reader is invited to compare it to the synthesized functor from Example 2.

$$\begin{array}{ll} \texttt{t} \mapsto (\lambda k.\lambda v.\, \texttt{T.t}\ v\ k) & \texttt{lookup} \mapsto (\texttt{fn t => fn x => T.lookup x t}) \\ \texttt{empty} \mapsto \texttt{T.empty} & \texttt{update} \mapsto (\texttt{fn t => fn x => fn y => T.update t (x,y)}) \end{array}$$

It is not hard to recover the SML expressions whose synthesis has been requested, from a solution to the corresponding sequent problem. Therefore, the task that our methods tackle is the search for solutions to a given sequent problem.

3.2 Search

Conceptually, the search can be split into four phases. However, in our implementation these phases interleave.

Phase 1: Goal-Directed Proof Search. The first phase consists of a goal-directed search in a classical logic sequent calculus. Such a search operates on a list of sequent problems — the list of *proof goals* — that share a common problem context. It attempts to find a substitution that simultaneously solves all of the goals. The rules of the sequent calculus decompose a proof goal into a list of new proof goals. At some point of the search process, proof goals are reduced to unification problems. This happens when both sides of a sequent problem contain an atomic formula under the same atomic predicate symbol. In this case, we reduce the problem to a unification problem by equating the types and terms left of $\vdash$ with the corresponding types and terms right of $\vdash$.

$$\mathcal{C};\ \begin{pmatrix} \Delta \cup \{(P\ \texttt{at}\ (ty_1, \ldots, ty_k))\ M_1 \ldots M_n\} \\ \vdash \Theta \cup \{(P\ \texttt{at}\ (ty'_1, \ldots, ty'_k))\ M'_1 \ldots M'_n\} \end{pmatrix} \quad \rightarrow \quad \mathcal{C};\ \left\{ \begin{array}{rcl} ty_1 & = & ty'_1 \\ & \ldots & \\ ty_k & = & ty'_k \\ M_1 & = & M'_1 \\ & \ldots & \\ M_n & = & M'_n \end{array} \right\}$$

The system of equations has two parts — a system of *type equations*, and a system of *term equations*. We are looking for a substitution that solves the type equations up to $\beta\eta$-equality, and the term equations up to observational equality in SML.

Phase 2: Unification up to $\beta\pi$-Equality. It seems hopeless to come up with a good unification procedure for SML observational equality, even for

the very restricted sublanguage of pure terms.[3] Instead, we enumerate solutions up to $\beta\pi$-equality, where π stands for the equational law for record selection. $\beta\pi$-equality is not sound in a call-by-value language. Therefore, the substitutions that get returned by $\beta\pi$-unification will later be applied to the original unification problems, and it will be checked whether they are solutions up to observational equality (see Phase 4). For $\beta\pi$-unification, we use a version of Huet's pre-unification procedure for simply typed λ-calculus [11]. A number of extensions were necessary to adapt $\beta\pi$-unification to our language:

First-Order Type Functions. Because our language has first-order type function variables, we need to use higher-order unification at the level of types. Type unification problems that come up in practice are simple and fall into the class of so-called *(higher-order) pattern problems* [14]. For these problems, unifiability is decidable. Moreover, these problems have most general unifiers and there is a terminating algorithm that finds them. Our implementation only attempts to solve type unification problems that are pattern problems.[4] If a type unification problem is not a pattern problem, it is postponed. If it can't be postponed further, the synthesizer gives up.

The synthesizer never attempts to solve a term disagreement pair if there are still unsolved type disagreement pairs in the system. This way, it avoids an additional source of non-termination that results from the fact that terms in a system with unsolved type disagreement pairs may be non-well-typed and, hence, non-terminating.

Polymorphism. Higher-order unification in the presence of polymorphic universal variables has been described in [16]. In addition to polymorphic universal variables, we also allow polymorphic existential variables. In order to handle them in a good way, we use an explicitly typed intermediate language in the style of Core-XML [8]. For higher-order unification in an explicitly typed language, the presence of type function variables is important. Type function variables facilitate the "raising" of fresh type variables that get introduced into the scope of bound type variables.

Record Types. We treat record types in a similar way as products types are treated in [2].

No Extensionality Rules. Most extensions of higher-order unification to more expressive languages than simply typed λ-calculus assume extensionality rules. Extensionality rules simplify unification both conceptually and computationally. Our unification procedure, however, does not use extensionality rules for functions or records, because only very limited forms are observationally valid in a call-by-value language like SML.

Phase 3: Enumerating Terms of Given Types. After unification, terms may still have free existential variables. These variables need to be replaced by terms that contain universal variables only. This is non-trivial, because the terms must have correct types. We need a procedure that enumerates terms of a given type in a given parameter context. By the Curry-Howard isomorphism, this is

[3] Pure terms are those SML expressions that we permit as arguments for predicates.
[4] In term unification, though, we also solve non-patterns.

achieved by a proof search in a fragment of intuitionistic logic. Because of the presence of polymorphic types, this fragment exceeds propositional logic. Our current experimental implementation uses a very simple, incomplete procedure. Eventually, this procedure will be replaced by an enumeration procedure based on a sequent calculus in the style of [10, 5].

Phase 4: Soundness Check for Observational Equality. After Phase 3, we have found substitutions that solve the original sequent problems up to $\beta\pi$-equality. However, $\beta\pi$-equality is not sound in a call-by-value language that has non-termination, let alone side-effects. For example, the following β-equality is not observationally valid, because the left hand side may not terminate for certain interpretations of f by SML expressions:

$$(\lambda x.\, 0)\,(f\,0) \;=\; 0 \tag{1}$$

The following β-equality is valid in functional SML but not in full SML including side effects, because it does not preserve the execution order:

$$(\lambda x.\, \lambda y.\, f\,y\,x)\,(g\,0)\,0 \;=\; f\,0\,(g\,0) \tag{2}$$

Because $\beta\pi$-equality is unsound in SML, the synthesizer applies the substitutions that have been discovered in Phases 1 through 3 to the original unification problems, and checks whether they solve them up to observational equality. For this check, it uses two different procedures depending on which option it has been invoked with. An option for functional SML guarantees correctness of the synthesized code in a purely functional subset of SML. An option for full SML guarantees correctness in full SML including side effects. Both checks are approximative and may reject $\beta\pi$-solutions unnecessarily. For full SML, the synthesizer uses $\beta_v\pi_v$-equality ("beta-pi-value-equality") [17]. $\beta_v\pi_v$-equality only allows β-reductions if the argument term is a syntactic value. Applications are not syntactic values. Therefore, Equation 1 is not a $\beta_v\pi_v$-equality. For functional SML, it uses a strengthening of $\beta_v\pi_v$-equality. Under this stronger equality, certain β-reductions that are not β_v-reductions are allowed in equality proofs. Roughly, β-reductions are allowed if the function parameter gets used in the function body. Equation 2 holds with respect to this stronger equality.

4 Related Work

Language Design. The AxML specification language is inspired by and closely related to EML (**E**xtended **ML**) [12]. In style, AxML specification axioms resemble EML specification axioms. However, EML axioms have some features that complicate the task of automatic component adaptation. For this reason, AxML puts certain restrictions on specification formulas, that are not present in EML. Most notably, whereas EML identifies specification formulas with expressions of the SML type `bool`, AxML has an additional type `prop` for specification formulas. As a result, AxML specification axioms come from a small, particularly well-behaved language, whereas EML axioms include all SML expressions of type `bool`. In particular, EML axioms may be non-terminating SML expressions, whereas subexpressions of AxML axioms always terminate.

Automatic Retrieval, Adaptation and Composition. A closely related field of research is the use of specifications as search keys for software libraries. In library retrieval, it is generally desirable to search for components that match queries up to a suitable notion of similarity, like for example type isomorphism. Adaptation is necessary to overcome remaining differences between retrieved and expected components. Because human assistance is assumed, semantic correctness of the retrieved components is not considered critical in library retrieval. As a result, much of the research in this field has focused on traditional type specifications [18, 20, 4, 1]. Work on library retrieval that is based on finer semantic specifications than traditional types includes [19, 6, 23]. All of these rely on existing theorem provers to do a large part of the work. Hemer and Lindsay present a general framework for lifting specification matching strategies from the unit level to the module level [9]. The Amphion system [21, 13] is interesting in that it does not only retrieve functions from libraries, but it also composes them. There is a major difference between Amphion and our system: Whereas in Amphion a pre-post-condition specification for a function of type $(\tau \to \tau')$ is expressed as $(\forall x : \tau.\exists y : \tau'.\ pre(x) \Rightarrow post(x, y))$, it is expressed in our system as $(\exists f : \tau \to \tau'.\forall x : \tau.\ pre(x) \Rightarrow post(x, fx))$.[5] As a result, the most important component of our synthesizer is a higher-order unification engine. Amphion, on the other hand, uses a resolution theorem prover for (constructive) first-order logic with equality.

5 Conclusion and Future Directions

We have enriched SML module interfaces by semantic specification axioms. We have assembled methods for automatically recognizing similarities between enriched interfaces and for synthesizing adapters that bridge incompatibilities. The use of uninterpreted predicate symbols prevents specification axioms from getting too detailed, so that our incomplete methods terminate quickly and successfully in many practical cases. The synthesized adapters preserve semantic correctness, as opposed to just type correctness. They may contain higher-order and polymorphic functions as well as parameterized type definitions.

In this paper, we have proposed to apply our methods for protecting evolving software systems against interface changes. We hope that, in the future, our methods can be extended and used in other ambitious applications:

Automatic Composition of Generic Library Functions. Standard libraries of modern programming languages often contain few, but very general, functions that can be composed and specialized in a lot of useful ways. Examples are I/O-primitives in standard libraries for Java or SML. The generality of such functions keeps the libraries both small and powerful. However, it can also make the functions hard to understand and difficult to use correctly. Programmers have to grasp the library functions in their full generality in order to be able to compose the special instances that they need. Even if they know in which modules of a library to look, naive programmers often find it hard to compose the

[5] This is how Amphion represents specifications internally. Amphion has an appealing graphical user interface.

library functions in the right way. Our hope is that, in the future, our methods can be extended to automatically compose library functions, based on simple semantic specifications given by the programmer. This kind of use puts a higher burden on the synthesis algorithms, because it assumes that the naive programmer only understands a limited vocabulary. He possibly uses simpler and less general atomic predicate symbols than the ones that are used in specifications of library functions. As a result, additional axioms are needed so that the synthesis tool can recognize the connection.

Automatic Library Retrieval. An interesting future direction is to extend and use our methods in automatic library retrieval based on semantic specifications. The difficulty is that libraries are large, and, thus, contain a large number of specification axioms. To deal with this difficulty, one could incorporate the AxML synthesizer as a confirmation filter into a filter pipeline, as proposed in [6].

References

[1] M. V. Aponte, R. D. Cosmo. Type isomorphisms for module signatures. In *8th Int'l Symp. Prog. Lang.: Implem., Logics & Programs, PLILP '96*, vol. 1140 of *LNCS*. Springer-Verlag, 1996.
[2] I. Cervesato, F. Pfenning. Linear higher-order pre-unification. In *Proc. 12th Ann. IEEE Symp. Logic in Comput. Sci.*, 1997.
[3] R. L. Constable, J. Hickey. Nuprl's class theory and its applications. Unpublished, 2000.
[4] R. Di Cosmo. *Isomorphisms of Types: From Lambda-Calculus to Information Retrieval and Language Design*. Birkhäuser, 1995.
[5] R. Dyckhoff, L. Pinto. A permutation-free sequent calculus for intuitionistic logic. Technical Report CS/96/9, University of St Andrews, 1996.
[6] B. Fischer, J. Schumann. NORA/HAMMR: Making deduction-based software component retrieval practical. In *Proc. CADE-14 Workshop on Automated Theorem Proving in Software Engineering*, 1997.
[7] C. Haack. *Foundations for a tool for the automatic adaptation of software components based on semantic specifications*. PhD thesis, Kansas State University, 2001.
[8] R. Harper, J. Mitchell. On the type structure of Standard ML. *ACM Trans. on Prog. Lang. and Sys.*, 15(2), 1993.
[9] D. Hemer, P. Lindsay. Specification-based retrieval strategies for module reuse. In *Australian Software Engineering Conference*. IEEE Computer Society Press, 2001.
[10] H. Herbelin. A λ-calculus structure isomorphic to Gentzen-style sequent calculus structure. In *Proc. Conf. Computer Science Logic*, vol. 933 of *LNCS*. Springer-Verlag, 1994.
[11] G. Huet. A unification algorithm for typed λ-calculus. *Theoret. Comput. Sci.*, 1(1), 1975.
[12] S. Kahrs, D. Sannella, A. Tarlecki. The definition of Extended ML: A gentle introduction. *Theoretical Computer Science*, 173, 1997.
[13] M. Lowry, A.Philpot, T. Pressburger, I. Underwood. Amphion: Automatic programming for scientific subroutine libraries. In *Proc. 8th Intl. Symp. on Methodologies for Intelligent Systems*, vol. 869 of *LNCS*, Charlotte, 1994.
[14] D. Miller. A logic programming language with lambda-abstraction, function variables and simple unification. *Journal of Logic and Computation*, 1(4), 1991.
[15] R. Milner, M. Tofte, R. Harper, D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
[16] T. Nipkow. Higher-order unification, polymorphism and subsorts. In *Proc. 2nd Int. Workshop Conditional & Typed Rewriting System*, LNCS. Springer-Verlag, 1990.
[17] G. D. Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theoret. Comput. Sci.*, 1, 1975.
[18] M. Rittri. Using types as search keys in function libraries. *J. Funct. Programming*, 1(1), 1991.
[19] E. R. Rollins, J. M. Wing. Specifications as search keys for software libraries. In K. Furukawa, ed., *Eighth International Conference on Logic Programming*. MIT Press, 1991.
[20] C. Runciman, I. Toyn. Retrieving reusable software components by polymorphic type. *Journal of Functional Programming*, 1(2), 1991.
[21] M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, I. Underwood. Deductive composition of astronomical software from subroutine libraries. In *The 12th International Conference on Automated Deduction*, Nancy, France, 1994.
[22] P. Wadler. Theorems for free! In *4th Int. Conf. on Functional Programming Languages and Computer Architecture*, 1989.
[23] A. M. Zaremski, J. M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Technology*, 6(4), 1997.