SOFTWARE VERIFICATION RESEARCH CENTRE

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 01-25

Specification-based Retrieval Strategies for Module Reuse

David Hemer and Peter Lindsay

August, 2001

Phone: +61 7 3365 1003
Fax: +61 7 3365 1533
http://svrc.it.uq.edu.au

Appears in the *Proceedings of Australian Software Engineering Conference (ASWEC'2001)*, D. Grant and L. Stirling eds., IEEE Computer Society Press, pp 235-243, August 2001

Specification-based Retrieval Strategies for Module Reuse

David Hemer and Peter Lindsay

Abstract

Formal specifications have been proposed as a basis for accessing reusable components from libraries, and various fine-grained specification-matching approaches have been developed to assist in searching libraries. Typically, however, the granularity of matching has been too fine for reuse to be effective. Compounding the problem is the fact that coarse-grained items usually require adaptation before reuse. This paper explains some of the problems and presents a generic solution to a key problem: adaptation of modules through parameter instantiation and subsetting. It shows how unit-matching strategies can be lifted to module level in a generic fashion.
Keywords: component reuse, retrieval, adaptation, specification matching.

1 Introduction

With the increasing interest in component-based technologies [3] comes renewed interest in formal specifications of components and development of libraries of reusable, formally specified components. Formal specifications allow components and their interfaces to be characterised concisely and precisely. Also, because formal specifications are machine parseable, they are ideal candidates as search keys [21]. In theory then, such specifications should make it easier to search component libraries, to check component properties, and to retrieve components for reuse.

In practice, however, it is not quite so easy. One reason is the multitude of different ways of specifying a component (or even of structuring its specification): it is rare that, in the absence of prior knowledge, a person searching a large library of components would formulate their search query in a way that exactly matches the possible solutions. Another reason is that the reusable component often requires adaptation for its new context, and (with some debatable exceptions) existing formal specification languages are generally weak in their support for adaptation of coarse-grained objects. These problems are explored in more detail below.

1.1 Specification-based search strategies

To date, most specification-matching and retrieval research has concentrated on the level of individual functional *units* within a formally specified component: e.g. function signatures in functional-language programming systems [13, 15]; definitions, axioms and theorems in theorem-proving systems [9]; and individual specification statements in formal software development environments. The typical approach is to search library modules (also referred to as *patterns* in this paper) unit-by-unit; when a matching unit is found, the module and the instantiation are returned, for manual adaptation – in other words, the formal specifications are being used as search keys only.

In practice however, reuse is more effective when it can be carried out at coarser-grained levels than units – namely, at the level of *modules* (structured collections of interdependent units) and above: e.g. function suites in functional-language programming systems [17]; whole theories or theory extensions in theorem-proving systems [4]; object classes in object-oriented programming systems [19]; and whole specifications in formal software development environments [21].

This paper is concerned with extending unit matching to module level and supporting multiple parallel queries. Searches can be narrowed significantly, and instantiations made more specific, by matching two or more units at a time, rather than unit by unit. This is analogous to enhancing the effectiveness of a web browser by allowing the use of multiple keywords.

1.2 Module adaptation

Searching is only one half of the retrieval problem: to reuse a module it can be just as difficult to indicate how the module should be adapted for its new context. Here it is more difficult to propose general solutions, since module adaptation is strongly dependent on the programming language being used.

This paper focuses on two key generic approaches to adaptation: parameter instantiation and extraction of functional subsets from modules. The first of these will be familiar to most readers, and relates to parametric polymorphism as supported by most modern formal-specification languages [11].

The motivation for exploring subsetting is the observation that library modules often contain far more units than would ever be used at one time. Consider for example the C "string" library [8] containing a variety of functions for manipulating strings. The library includes functions for comparing strings, concatenating strings, calculating the length of a string etc. Instead of including the entire library, it is often desirable to include only those parts of the library required by the application. In the context of formal languages, restricting a module to a necessary subset can result in less proof. Subsetting can also result in the ability to generate more efficient code, and reduce cluttering of the application code.

In practice, modules generally have richer structure than simply being a collection of mutually dependent units, as they are treated in this paper. For example, many object oriented languages have a notion of submodules, with importation, inheritance, and so on. Because of the diversity of such structures it is impossible to give general solutions, but the subset-extraction solution proposed here should generalise well in most cases.

1.3 Outline of this paper

Section 2 surveys a number of existing examples where specification matching is used to retrieve fine-grained components. From these examples a general framework for adapting and retrieving fine-grained components is defined.

Section 3 extends the framework defined in Section 2 to include coarser-grained components. A module adaptation technique, referred to as subsetting, and a general model for adapting and matching coarse grained components are formulated.

Section 4 describes three different strategies for matching modules. Section 5 gives an example application of the module reuse framework.

A KIDS notation [16] is used for many of the examples. The general framework presented in this paper has been applied to extend the CARE toolset [6]. CARE consists of methods and tools for developing formally verified code from high-level formal specifications. Included are a number of reusable modules which capture commonly used algorithms, data structures or lower level coding constructs. By applying the general framework described in this paper, we have extended the support for adaptation of modules, as well as developing a retrieval tool — based on the module matching framework — to assist the software engineer in semi-automating the development process.

2 Matching fine-grained components

2.1 Existing approaches

2.1.1 Signature matching

In functional programming languages, function *signatures* describe the types and numbers of inputs and outputs. *Signature matching* [20], uses the signature as a query in searching the library. A library of functions can be searched by giving the signature of a desired component (the query) and attempting to match this signature against the signatures of the library components (the patterns).

Example 2.1 Suppose the user wants to implement a function for inserting an integer in a (sorted) set of integers. In ML [17], the signature for such as function is:

$$\mathsf{insert\_int} : \mathsf{int} \times \mathsf{int\ set} \rightarrow \mathsf{int\ set}$$

Assuming the ML library contains the function insert for inserting an element in a set, with the following signature:

$$\mathsf{insert} : \mathsf{a} \times \mathsf{a\ set} \rightarrow \mathsf{a\ set}$$

where *a* is a type parameter. Then a signature-matching based search tool, using insert_int as the query would

match the library function insert, by instantiating the parameter $a \rightsquigarrow int$. □

The shortcoming of this approach is that while the technique successfully matches the desired library function, in general many other functions will also have a signature that matches the query.

2.1.2 Specification matching

Specification matching [21, 14, 12, 7], goes some way towards addressing the major shortcoming of signature matching, by specifying functions more precisely using pre- and post-conditions. Matching involves comparing the pre- and post-conditions of the query against those of the library functions.

Zaremski and Wing [21] describe a variety of ways that a function can satisfy a query, where the function and query are both specified using pre- and postconditions. *Exact pre/post* is where the corresponding preconditions and postconditions are equivalent; *plug-in match* is where the precondition of the library component is weaker than that of the query, and the postcondition of the library component is stronger; and *exact predicate* is where the conjunction of the precondition and postcondition for the query and library component are equivalent.

Example 2.2 Consider the four KIDS-like functions [16] shown in Fig. 1. Each function returns the first integer in a list of numbers, provided certain conditions are satisfied. The first function hd1 includes a precondition stating that the list contains at least one element and returns the element with index "1" in the list (i.e., the first element). Function hd2, has a precondition stating that the list is non-empty and returns the head of the list. Function hd3 returns the head of a non-empty list, or returns "0" for an empty list. Function hd4 has a precondition stating that the list contains at least two elements and returns the element with index "1".

The pre- and post-conditions of the functions hd1 and hd2 are logically equivalent, and therefore match using all three of the strategies described above. The function hd3 has a weaker pre-condition and stronger post-condition than the functions hd1, hd2 and hd4, therefore each of the these functions used as queries will match the library function hd3 using plug-in match. The conjunction of the pre- and post-conditions in hd1, hd2 and hd3 are logically equivalent, and therefore these functions match using exact predicate. □

function hd1(x: seq(integer)): integer
  where $\#x > 0$
  returns $\{z \mid x(1) = z\}$.

function hd2(x: seq(integer)) : integer
  where $x \neq \langle\,\rangle$
  returns $\{z \mid z = \mathit{head}(x)\}$.

function hd3(x: seq(integer)) : integer
  where true
  returns $\{z \mid z = \mathit{if}\ z \neq \langle\,\rangle\ \mathit{then}\ \mathit{head}(x)\ \mathit{else}\ 0\}$.

function hd4(x: seq(integer)) : integer
  where $\#x > 2$
  returns $\{z \mid x(1) = z\}$.

Figure 1: Functions for returning the head of a list, in KIDS notation

2.1.3 Rule matching

Matching of logical inference rules occurs in theorem provers (e.g., Isabelle [1]). The query will typically be formed from the proof goal — it may be the entire goal, a subpart of the goal, or one or more of the local assumptions. The patterns are definitions and theorems from theories currently in scope. Both query and pattern are predicates, possibly containing higher-order parameters. The pattern is said to satisfy the query if there is some instantiation of formal parameters in the two predicates, such that the instantiated predicates are equal up to renaming of bound variables (i.e. *alpha-equivalent*).

Example 2.3 Suppose the user, in Isabelle, is required to prove that removing numbers less than 5 from a list of numbers results in a list with length equal to or less that the original list, i.e.,

```
length (filter (λ x • x ≥ 5)
    [a,b,c]) ≤ length [a,b,c]
```

To do this the user can apply the rule `length_filter`, stating that the length of a filtered list is less than or equal to the length of the original, i.e.:

```
length (filter P xs) ≤ length xs
```

The rule includes the first-order parameter *xs*, representing the list of elements being filtered, and the higher-order parameter *P*, representing relation over the list.

In applying the rule, the Isabelle theorem prover attempts to match the rule against the subgoal. It succeeds by instantiating the parameters $xs \rightsquigarrow [a, b, c]$ and $P \rightsquigarrow (\lambda x \bullet x \geq 5)$. □

2.2 A generalised model

This section defines a general framework for adapting and retrieving units. The Z specification language [18] is used to define the framework, with certain concepts left underdefined. The framework is built up gradually, starting with unit-level "primitives" (generic data types and functions, and a minimal set of assumptions about their properties) upon which higher-level constructs will be defined.

2.2.1 Units and queries

First, we introduce generic types to represent the components retrieved and used by the user (also referred to as *patterns*), and the queries used to search for these components. These are modeled in Z using "generic" types:

$$[Unit, UnitQuery]$$

Units are those components that reside in the library. Examples of units include functions in ML [17] and C [8]; axioms and theorems in Isabelle [1]; state and operational schemas as well as axiomatic definitions in Z [18]; functional statements in KIDS [16]; and abstract state machines in B [2]. A unit query encapsulates the user's requirements. Quite often the query contains a subset of the information associated with a unit. For example, for signature matching, the query only contains information about the prospective function's type signature. Similarly, for specification matching the query includes the pre- and post-conditions of the required function, but not any implementation details.

Next, we assume there is a relationship *Satisfies* which captures the notion of a unit satisfying the requirements expressed in a unit query.

$$| \; Satisfies : Unit \leftrightarrow UnitQuery$$

For example, for signature matching, a function satisfies a query, if the signature of the function is equal to the signature given by the query. For more examples, consider specification matching: Section 2.1.2 describes three *satisfies relations*, as used in the exact pre/post, plug-in match and exact predicate strategies. For exact pre/post a function satisfies a query if the corresponding pre- and post-conditions are logically equivalent. For exact predicate, a function satisfies a query if the conjunction of the pre- and post-conditions are logically equivalent. For plug-in match, a function satisfies a query if the precondition of the function is weaker than that of the query, and the post-condition of the function is stronger than that of the query. For the exact pre/post and exact predicate strategies the satisfies relation is an equivalence relation (i.e., the relation is reflexive, transitive and symmetric). However for the plug-in match strategy, the satisfies relation is not an equivalence relation (it is not symmetric), instead the relation is a pre-order.

2.2.2 Adaptation

By making units adaptable, a library of units can solve a wider range of problems than a similar sized library of rigid units. This has the flow-on effect of decreasing the library and subsequently making the search space smaller. Let the following Z type represent the possible ways a unit can be adapted:

$$[UnitAdapt]$$

and let the following function represent the effect of applying an adaptation:

$$| \; adapt : Unit \times UnitAdapt \rightarrow Unit$$

Unit adaptations are dependent on the application language; however there are general classes of adaptations that can be applied to most languages. The majority of current approaches only consider instantiation of formal parameters, e.g., Isabelle includes instantiation of higher-order parameters. However our framework supports an arbitrary set of adaptations, restricted only by the proviso that the adaptation returns a unit obeying the syntactic and semantic constraints of the application language. As well as instantiation of formal parameters, other examples of adaptations of functions include renaming of function, type and variable names, and reordering of the inputs of the function.

Example 2.4 Consider the function fun, parameterised over the relations P and Q:

function fun(x:X,y:Y):Z
  where $P(x, y)$
  returns $\{z \mid Q(x, y, z)\}$

This function can for example be adapted by: renaming the function name fun to div; instantiating the types X, Y and Z by integer; renaming the variables x, y and z to a, b and c; swapping the input variables; and instantiating the parameters $P \rightsquigarrow \lambda\, a, b \bullet a \neq 0$ and $Q \rightsquigarrow \lambda\, a, b, c \bullet c = b \operatorname{div} a$. The resulting adapted function is:

function div(b:integer,a:integer):integer
  where $a \neq 0$
  returns $\{c \mid c = b \operatorname{div} a\}$

Notice that the result is a valid (syntactically and semantically correct) function. □

2.2.3 Matching

Next we assume there is a 3-place predicate *matches* which represents the fact that a given unit matches a given unit query via a given adaptation:

$$\begin{array}{|l} \mathit{matches} : \mathbb{P}(\mathit{Unit} \times \mathit{UnitQuery} \times \mathit{UnitAdapt}) \\ \hline \forall\, u : \mathit{Unit};\ q : \mathit{UnitQuery};\ a : \mathit{UnitAdapt} \bullet \\ \quad \mathit{matches}(u, q, a) \Leftrightarrow \mathit{adapt}(u, a)\ \mathit{Satisfies}\ q \end{array}$$

Note that a unit may match a query in many different ways.

Example 2.5 Consider query of the form:

function Query(x:integer,y:integer):integer
  where $x + y > 0$
  returns $\{z \mid z = x + y + 1\}$

and a library function

function Pattern(a:X,b:Y):X
  where $P(a, b)$
  returns $\{c \mid c = f(a, g(b))\}$

where $f : X \times Y \rightarrow X$, $g : Y \rightarrow Y$ and P are formal parameters. Query matches Pattern by: renaming the types X and Y to integer; renaming the function name Pattern to Query; renaming the variables a, b and c to x, y and z; and instantiating $f \rightsquigarrow \lambda\, x, y \bullet x + y$, $g \rightsquigarrow \lambda\, y \bullet y + 1$ and $P \rightsquigarrow \lambda\, x, y \bullet x + y > 0$. Alternatively, they can be matched by instantiating the parameters to $f \rightsquigarrow \lambda\, x, y \bullet x + y + 1$ and $g \rightsquigarrow \lambda\, y \bullet y$ instead. □

Finally, we assume there is a function *match*, that returns a set of matches between a query and unit. The defining property of the *match* function states that any matches between a unit and query must satisfy the *matches* relation (we make no assumptions about completeness of the function however):

$$\begin{array}{|l} \mathit{match} : \mathit{Unit} \times \mathit{UnitQuery} \rightarrow \mathbb{F}\, \mathit{UnitAdapt} \\ \hline \forall\, u : \mathit{Unit};\ q : \mathit{UnitQuery};\ a : \mathit{UnitAdapt} \bullet \\ \quad a \in \mathit{match}(u, q) \Rightarrow \mathit{matches}(u, q, a) \end{array}$$

This completes our description of the primitives of the framework. Note that the framework is very general and covers all of the examples in Section 2.1.

3 Matching coarse-grained components

Many languages employ module-like structures to store a number of related units together in a library. However the majority of specification matching approaches are restricted to individual units. This section describes how the unit matching primitives in Section 2.2 can be lifted to the module level.

3.1 Modules

The framework for matching units is extended to handle coarser-grained components, referred to here as *modules*. Examples of modules include theories in Isabelle; classes in Object-Z and C++; templates in CARE; and packages in Ada. For generality, no assumptions will be made about the structure of modules, other than that there is a finite set of units associated with the module which take part in retrieval and adaptation. (Such units would typically reside in the module's interface: e.g., declarations in an Ada package header). To maintain generality, we shall not formally model modules here; in particular no assumptions shall be made about the structuring of modules.

[*Module*]

To access this set of units we assume that the function *unitsOf*, which returns the set of units contained within a module, has been provided.

$$unitsOf : Module \rightarrow \mathbb{F}\, Unit$$

The exact nature of this function will depend on the particular modules in question. For a flat module, *unitsOf* will just return the set of units contained in the module. However for hierarchically structured modules *unitsOf* may represent a recursive function which returns the set of units contained within the nested modules. Similarly, in object-oriented programming, the function may need to traverse the inheritance structure.

3.2 Adapting modules

The two general forms of module adaptation considered in this paper concern adapting individual units, and module *subsetting*.

Subsetting is useful when the user only requires some of the functionality offered by a module. Subsetting returns a submodule, itself a module, obeying the same syntactic and semantic constraints of its parent. Properties that applied to the parent module, should also hold for any submodules. For example if a module is self-contained (i.e., any unit used in the module is also defined in the module), then any submodules should also be self-contained. In this case, when the user nominates a subset of the module, the adaptation tool adds any further units from the module to ensure that a self-contained submodule is returned.

The exact details of the mechanisms required to extract the submodules are highly application-language specific. Consider the following two examples.

Example 3.1 Templates in CARE [6] have a flat structure, in that they do not inherit other templates, and they are self-contained, in that any units used in the template are also declared in the template. Subsetting in this case is fairly straightforward — the user specifies a subset of the units in the template, and the adaptation tool calculates the closure of this set, by ensuring that any unit used in the subset is also declared in the subset. □

Example 3.2 A theory in Isabelle consists of a set of constructs (such as rules, definitions, type declarations etc.), as well as a listing of inherited theories (the so-called *parents*). The parent theories can in turn inherit other theories; the *ancestors* of a theory are the union of the parent theories and ancestors of the parents.

Extracting a sub-theory of an Isabelle theory would firstly involve restricting the constructs in the current theory to a subset. Next it may be possible to restrict the ancestors to a subset of the original ancestors. To do this, we require knowledge of fine-grained dependencies between theories (i.e. at the level of individual theory constructs).

Limiting a theory to a subset can result in a theory with a smaller search space, that requires less memory to save, and is quicker to load. □

We define a *module adaptation* to consist of a single unit adaptation and a set of module units. The unit adaptation describes how the individual units within the module are adapted; a single adaptation is used to ensure that adaptations are applied consistently throughout the entire module — for example it ensures that parameters are instantiated to the same value throughout the module. The set of module units describes, for the purpose of module subsetting, what module units to include.

$$ModAdapt == UnitAdapt \times \mathbb{F}\, Unit$$

3.3 Module matching

A query for matching modules consists of a set of unit queries.

$$ModQuery == \mathbb{F}\, UnitQuery$$

This is a useful way of factoring a problem into individual requirements. For example, suppose the user is searching for a module that includes functions for reversing a list, concatenating two lists, and calculating the length of a list. Such a requirement can be broken down into three individual requirements in this framework. Section 4 illustrates that breaking the query down into individual requirements allows the user to search for modules that satisfy only some of the requirements (i.e. SOME-match) or exactly one of the requirements (i.e. ONE-match).

The retrieval tool is based on *matching* a set of unit queries against a module (so-called *module matching*). Section 4 describes three different strategies for matching modules: ALL-match, SOME-match and ONE-match.

4 Module matching strategies

4.1 ALL-match

The first module matching strategy described is ALL-match, in which each of the unit queries must be matched against a module unit.

$$
\begin{array}{|l}
matches_{all} : \mathbb{P}(Module \times ModQuery \times ModAdapt) \\
\hline
\forall\, m : Module;\ qs : ModQuery;\ a : ModAdapt \bullet \\
\quad matches_{all}(m, qs, a) \Leftrightarrow \\
\qquad \forall\, q \in qs \bullet \exists\, u : Unit \bullet u \in unitsOf(m) \wedge \\
\qquad\quad matches_{unit}(u, q, \pi_1 a) \wedge u \in \pi_2 a
\end{array}
$$

This search strategy is useful when the user requires a number of units with one or more shared requirements (e.g., a number of functions for manipulating an abstract data type that are based on the same underlying type). By specifying the individual requirements in separate unit queries, and searching the library using the ALL-match strategy, only modules that satisfy each of the requirements are returned.

Suppose, for example, the user wishes to find implementations for a number of list manipulating primitives. Suppose the library contains abstract data types for manipulating lists based on three different underlying representations — linked lists, doubly linked lists and arrays. In searching the library for functions which implement the user's primitives, we need to ensure that the implementations returned share the same underlying representation, i.e., all of the functions should come from the same ADT. To do this the required functions can be specified in a single (module) query, and a search conducted using the ALL-match strategy, such that all of the functional requirements are satisfied by a single library module.

4.2 SOME-match

Matching *some* unit queries against a module is a relaxation of the stricter ALL-match strategy given in the previous section. Of interest here are modules which containing units which match a nonempty subset of the query set. This will include the set of matches formed by the ALL-match strategy. As would be expected, this kind of search is less precise that the previous one, and results in more matches being found. In contrast to ALL-match where an increase in the number of unit queries results in a decrease in matches; for the SOME-match an increase in the number of unit queries results in an increase in the number of matches. For this reason the user needs to be somewhat judicious in the number of units in the query given for this kind of match.

$$
\begin{array}{|l}
matches_{some} : \mathbb{P}(Module \times ModQuery \times ModAdapt) \\
\hline
\forall\, m : Module;\ qs : ModQuery;\ a : ModAdapt \bullet \\
matches_{some}(m, qs, a) \Leftrightarrow \\
\quad \exists\, ss \subseteq qs \bullet ss \neq \varnothing \wedge matches_{all}(m, ss, a)
\end{array}
$$

Consider development of programs using a step-wise refinement approach as used in systems such as CARE [6] and B [2]. The starting point is a specification of the desired component, represented by one or more unit specifications. At each stage in the development one or more of the specified-only units are implemented by other (more concrete) units, with these new units added to the program. The process continues until all units in the program are implemented. To semi-automate each stage in the development a search of a library of predefined modules could be done to find units satisfying the specifications of the unimplemented units in the program. The search query is the set of specified-only units in the program. Attempting to match each of the query units (using the ALL-match strategy) is too strict a requirement in this case. Matching just some of the query units (i.e. using the SOME-match strategy) is sufficient, since the step-wise refinement approach only requires that at least one unit is implemented at each stage.

A similar situation occurs with theorem provers, where a set of conditions need to be discharged. At each stage in the proof one or more of these conditions must be proved, with each proof step possibly introducing new conditions. To automate the proof process a search tool could be used to find assertions from a library of theories which satisfy one or more of the conditions. The SOME-match strategy again could be used to find theories which satisfy some of these conditions.

4.3 ONE-match

A third strategy for module matching is to match *exactly one* query unit; this strategy is referred to as ONE-match. This can be thought of as being at the other end of the

spectrum to matching all units. Note that the subset description part of the module adaptation contains exactly one unit.

$$
\begin{array}{|l}
matches_{one} : \mathbb{P}(Module \times ModQuery \times ModAdapt) \\
\hline
\forall\, m : Module;\ qs : ModQuery;\ a : ModAdapt \bullet \\
\quad matches_{one}(m, qs, a) \Leftrightarrow \\
\qquad \exists\, q \in qs;\ u : Unit \bullet u \in unitsOf(m) \wedge \\
\qquad\quad matches_{unit}(u, q, \pi_1 a) \wedge \pi_2 a = \{u\}
\end{array}
$$

The ONE-match strategy enables the user to include a number of alternate unit queries in order to find a single desired unit. This could be used when there are a number of equivalent ways of specifying a unit; for example the user might desire a unit for manipulating a list s, with a precondition stating that the list is nonempty — such a precondition could be given as either $\#s \neq 0$ or $s \neq \langle\,\rangle$. In the case where logical equivalence is used, a single query, using either of the pre-conditions could be used. However when the satisfies relation is stronger — alpha-equivalence for example — it would be necessary to formulate two queries with the preconditions given above, but only one would be expected to match at a time.

In CARE, the specifications of fragments can contain a number of branches, each returning a different type of result. The retrieval tool for CARE uses the ONE-match strategy to search for these so-called "branching fragments". The query for searching for such a fragment, consists of multiple unit queries representing the different ways of ordering the specification branches. This module query is generated automatically by the tool after the user has entered any one of the possible branch permutations. Note that the CARE retrieval tool could have alternatively handled this problem by including reordering of specification branches as an adaptation (of fragments). However the point to be made here is that the framework allows for this flexibility.

Another application of the ONE-match strategy is in a theorem prover, where the goal is a disjunction of conditions

```
c1 ∨ c2 ∨ ... ∨ cN
```

To prove such a goal, it is sufficient to find a rule matching exactly one of the conditions. The implementation of the matching function employed would involve lazy evaluation of the matches, in that the match function would terminate once a match is found.

5 Applying the framework

Within a functional programming language, related functions that collectively implement an algorithm, or manipulate a particular data structure could be grouped together using a modular mechanism (cf. *templates* in CARE). Such an idea could easily be applied in KIDS — the following module is actually a CARE template that has been translated into KIDS notation [10]. Consider the module given in Fig. 2 consisting of a set of functions, collectively implementing a generic accumulator algorithm.

```
function Main(s: seq(X)) : Y
    returns {z | z = m(s)}
= if s = ⟨ ⟩
    then Base
    else Step(Head(s),Main(Tail(s)))
function Step(x: X, y:Y) : Y
    returns {z | z = f(x,y)}
function Base(): Y
    returns {z | z = b}
function Head(s: seq(X)) : X
    where #s > 0
    returns {z | z = head(s)}
function Tail(s : seq(X)) : seq(X)
    where #s > 0
    returns {z | z = tail(s)}
```

Figure 2: An accumulator module in KIDS

The module is parameterised over the types X and Y, functions $m : \mathrm{seq}\, X \rightarrow Y$ and $f : X \times Y \rightarrow Y$, and the constant $b : Y$ such that the following conditions are satisfied:

$$m(\langle\,\rangle) = b$$
$$m(\langle h \rangle \frown t) = f(h, m(t))$$

For an empty list, the accumulator function m returns the base element b. For a non-empty list, m applies the step function f to the head of the list and the result of applying the accumulator function recursively to the remainder of the list.

In this section a program for summing the elements in a list of integers is developed. Initially a top-level specification of the problem, in terms of the function

sum : seq $\mathbb{Z} \rightarrow \mathbb{Z}$ is given:

$$sum(\langle\,\rangle) = 0$$
$$sum(\langle h \rangle \frown t) = h + sum(t)$$

For a non-empty list, *sum* adds the head of the list to the sum of the remainder of the list. The sum of an empty list is defined to be zero.

A specification of the KIDS function Sumlist is given, which represents summing the elements of a list. The top-level specification indicates that a function for adding two integers, and a function for returning zero are also required. These two requirements are represented in the function queries Add and Zero.

function Sumlist(s: seq(integer)) : integer
returns $\{z \mid z = sum(s)\}$

function Add(n,m:integer) : integer
returns $\{z \mid z = n + m\}$

function Zero() : integer
returns $\{z \mid z = 0\}$

With the above functions as a search query, and using the ALL-match strategy, a match with the accumulator module is returned. Sumlist matches Main, Add matches Step and Zero matches Base. The query matches the module by renaming the function names Main to Sumlist, Step to Add and Base to Zero. The parameters are instantiated as $m \rightsquigarrow \lambda s \bullet sum(s)$, $base \rightsquigarrow 0$, $f \rightsquigarrow \lambda a, b \bullet a + b$, $X \rightsquigarrow integer$ and $Y \rightsquigarrow integer$. The conditions on the module parameters must hold for any values that the parameters are instantiated to; in this case the instantiated conditions correspond to the definition of *sum*, and so are trivially satisfied.

There are now four unimplemented functions. Suppose these functions are used as the next search query, this time using the SOME-match strategy, indicating that only some of the functions need to be implemented. Assuming that there exists a module containing primitives for manipulating lists, and another module containing primitives that manipulate integers. Then the query matches the functions Head and Tail against the list module, or match the functions Add and Zero against appropriate functions in the integer module. The development is completed by applying the results of each of these matches in turn.

6 Conclusions

Formal specifications allow components and their interfaces to be characterised concisely and precisely. Also, because formal specifications are machine parseable, they are ideal candidates as search keys. This paper discusses strategies for retrieving formally specified software components from reusable libraries. The solutions presented are quite general and apply across a range of different component types and formal specification approaches. They presume only that unit-level matching algorithms are available, and show how such algorithms can be lifted to module level to yield a range of different retrieval strategies.

Strategies are presented for matching all, some and one of multiple unit queries. The strategies are formally specified; for implementation details and and sketches of proofs of correctness the reader is referred to David Hemer's thesis [5]. It is shown that the algorithms are complete, in the sense that they return all "most general" matches, modulo completeness of the underlying unit-matching algorithms.

The design of a general interactive search tool is outlined in the thesis, and its use illustrated on specific retrieval support tools. A prototype of the search engine has been implemented in Prolog and applied to yield a template-library browser and retrieval tool for the CARE tool-set.

The approach and solution are expected to be of interest to tool-builders for applications involving libraries of reusable modules, such as formal theories in theorem-proving environments and formally specified components in software development support environments.

References

[1] Isabelle home page. `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/index.html`.

[2] J.-R. Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[3] A. W. Brown, editor. *Component-Based Software Engineering*. IEEE Computer Society, Carnegie Mellon University, Software Engineering Institute, 1996.

[4] D. Good. Reusable problem domain theories. Technical Report 31, ICSCA, University of Texas at Austin, 1982.

[5] D. Hemer. *A Unified Approach to Adapting and Retrieving Formally Specified Components for Reuse*. PhD thesis, Department of Computer Science and Electrical Engineering, University of Queensland, April 2000.

[6] D. Hemer and P. Lindsay. The CARE toolset for developing verified programs from formal specifications. In O. Frieder and J. Wigglesworth, editors, *Proceeding of the Fourth International Symposium on Assessment of Software Tools*, pages 24–35. IEEE Computer Society Press, May 1996. SVRC TR 95-52.

[7] J.-J. Jeng and B. Cheng. Specification matching for software reuse: A foundation. In *Proc. of ACM Symposium on Software Reuse*, pages 97–105, April 1995.

[8] A. Kelley and I. Pohl. *A Book on C*. Benjamin Cummins, third edition, 1995.

[9] P. Lindsay. A survey of mechanical support for formal reasoning. *Software Engineering Journal*, 3(1):3–27, January 1988.

[10] P. Lindsay and D. Hemer. A template-based approach to construction of verified software. Technical Report 96-23, Software Verification Research Centre, 1996.

[11] R. Milner. A theory of type polymorphism in programming. *JCSS*, 17:348–375, 1978.

[12] D. Perry and S. Popovich. Inquire: Predicate-based use and reuse. In *Proceedings of the 8th Knowledge-Based Software Engineering Conference*, pages 144–151, September 1993.

[13] M. Rittri. Using types as search keys in function libraries. In *Proceedings of the Fourth International Conference on Functional Programming and Computer Architecture*, pages 174–183. ACM Press, 1989.

[14] E. Rollins and J. Wing. Specifications as search keys for software libraries. In K. Furukawa, editor, *Eighth International Conference on Logic Programming*, pages 173–187. MIT Press, 1991.

[15] C. Runciman and I. Toyn. Retrieving re-usable software components by polymorphic type. In *Proceedings of the Fourth International Conference on Functional Programming and Computer Architecture*, pages 166–173. ACM Press, 1989.

[16] D. Smith. KIDS: A semiautomatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, September 1990.

[17] S. Sokolowski. *Applicative High Order Programming*. Chapman and Hall, 1991.

[18] J. Spivey. *The Z Notation: a Reference Manual*. Prentice-Hall, New York, 1989.

[19] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, second edition, 1991.

[20] A. Zaremski and J. Wing. Signature matching: a tool for using software libraries. *ACM Transactions on Software Engineering and Methodology*, 4(2):146–170, April 1995.

[21] A. M. Zaremski and J. Wing. Specification matching of software components. In *Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1996.