Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 187 (2007) 173–188
www.elsevier.com/locate/entcs

Semi-Automated Component-Based Development of Formally Verified Software

David Hemer[1]

School of Computer Science, The University of Adelaide, Adelaide, South Australia, 5005, Australia

Abstract

A number of formal approaches to component-based software development have been proposed, based on the idea of using formal specifications as a basis for retrieval. These approaches provide good recall and precision when searching for components. More recently, the problem of component adaptation has begun to be addressed, in recognition of the fact that a library component will rarely meet the needs of the user exactly. However the main weakness of the current approaches is they only cater for a single adaptation step.
In reality, we typically need to apply some combination of adaptation steps. In this paper we present a collection of search tactics, which allow us to combine a sequence of matching and adaptation commands into a single step. The tactics are presented in a general manner, with the intention that they could be applied to a variety of different formal-based approaches to CBSD. We illustrate the use of the search tactics using a simple example.

Keywords: Components; adaptation; formal methods

1 Introduction

For safety critical software a component-based software development (CBSD) approach, using pre-verified library components, can bring savings, particularly in the cost associated with verification. However a component-based approach is only viable when the overall effort in reusing components is significantly less than the effort in developing (and verifying) the software from scratch [13].

Like other CBSD approaches, there are many challenges that need to be addressed before any real savings can be made [3]. These challenges include locating suitable components and adapting them to meet the specific needs of the software engineer. Traditionally, component retrieval approaches were based on text or keyword based matching. However due to the ambiguity and imprecision associated

[1] Email: david.hemer@adelaide.edu.au

doi:10.1016/j.entcs.2006.08.051

with natural language specifications, such retrieval methods have poor precision and poor recall.

To counter these problems, methods for locating components in a library, based on matching formal component specifications have been developed [12,8,15]. More recently, methods for adapting formally specified have been proposed [10,9].

To produce a close synergy between matching and adaptation, we implement adaptation strategies using generic library components [5]. In effect finding a suitable solution to a user requirement involves a sequence of matching steps. During each step library components are composed and adapted until an exact fit with the user requirements is obtained.

In practice this approach often involves a number of mundane and tedious steps. In this paper we define search tactics, which can be used to combine multiple matching and adaptation steps into a single step. The idea is analogous to the use of tactics in interactive theorem provers, and indeed we draw much inspiration from this work. With these search tactics we aim to automate common component adaptation steps, allowing the software engineering to concentrate on more important design decisions.

In this paper we define a collection of search tactics used to semi-automate component adaptation and retrieval. It is our intention that the tactics should be applicable to any approach that uses formal-based matching and adaptation techniques. We therefore begin in Section 2 by defining a generic model for matching and adapting formally specified components. We consider individual units and modular components separately.

In Section 3 we define the general form of search tactics. We also define a collection of basic search tactics from which more complex tactics will be built. In Section 4 we define a collection of tacticals, used to combine basic tactics to build more complex tactics. In Section 5 we illustrate the use of the search tactics, by looking at a simple example using the CARE language and tools. Section 6 contains a comparison of our approach to other related work.

2 Logic-based selection of components

In this section we present a generic framework for matching and adapting formally specified components. The framework generalises existing approaches to specification matching, capturing the general notion of matching components with respect to adaptations of the components. The framework used here is based on a generic framework presented in an earlier paper [6]; however there are several changes which are noted in the text below. We model the framework using the Z specification language [14].

The generic framework considers components at two separate levels of granularity. At the first level we consider individual units, which correspond to functions, types, theorems, etc. At the second level we consider modular components,

Matching strategy	Condition
Exact Match	$(Q_{pre} \Leftrightarrow S_{pre}) \wedge (Q_{post} \Leftrightarrow S_{post})$
Plug-in Match	$(Q_{pre} \Rightarrow S_{pre}) \wedge (S_{post} \Rightarrow Q_{post})$
Guarded Plug-in Match	$(Q_{pre} \Rightarrow S_{pre}) \wedge$ $((S_{pre} \wedge S_{post}) \Rightarrow Q_{post})$
Satisfies Match	$(Q_{pre} \Rightarrow S_{pre}) \wedge$ $((Q_{pre} \wedge S_{post}) \Rightarrow Q_{post})$

Table 1
Semantic-based specification matching strategies

which consist of multiple units. This separation allows us to focus on matching and adaptation techniques relevant to the particular level of granularity. In developing matching methods for individual units, we do not need to consider how a collection of units will be matched. Similarly, when developing methods for matching modules, we can assume there is a method for matching the individual units.

2.1 Unit matching

We begin by considering the basic individual units that are contained within modular library components and programs. Examples of units include functions, types, theorems etc. We model these units using a generic type, representing a set of values that are not further defined.

$$[Unit]$$

We make no assumptions about the internal structure of units, but in the remainder of the section we describe a number of functions and relations that must be defined for each kind of unit.

We assume there is relationship between a pair of units, defining what it means for the first unit to satisfy the second. Generally the satisfies relationship will apply to the specification part of units.

$$Satisfies : Unit \leftrightarrow Unit$$

Table 1 shows some examples of satisfies relationships for some of the more commonly used specification matching techniques for function-like units that are specified using pre- and post-conditions. A more comprehensive collection of matching relationships is given by Zaremski and Wing [15]. In the table the library unit specification is represented by S, and the query specification by Q. The subscripts *pre* and *post* refer to the pre-condition and post-condition respectively.

Exact match succeeds when the preconditions of Q and S are equivalent, and when the postconditions of Q and S are equivalent. *Plug-in match* succeeds when the precondition of the library component is weaker than that of the query, and the postcondition of the library component is stronger than that of the query. *Guarded plug-in match* is based on plug-in match, but adds the precondition of the library component to the post-condition relation. *Satisfies match* is similar to guarded plug-in match [11], but uses the precondition of the query component instead of the library component to guard the post-condition relation. Notice that as we go down the table the matching relationships become weaker, so a plug-in match will also be an exact match for example.

By making units adaptable a library of components can be applied in different ways to solve a variety of problems. We define a generic type to represent the ways in which a unit can be adapted:

$$[Adapt]$$

We will typically only be interested in supporting relatively straightforward adaptations at this point. By straightforward, we mean adaptations that can be readily computed using automated techniques such as unification of unit specifications. Examples of such techniques include identifier renaming (e.g., for variables and type names), and higher-order variable instantiation. In cases where there are different kinds of adaptations that can be applied, we would model *Adapt* as a tuple.

We define the function *adapt* that applies an adaptation to a unit to yield a new unit. We also define the trivial adaptation, *trivAdapt*, which leaves units unchanged when applied to the *adapt* function, together with the function *merge*, which merges two adaptations.

$$\begin{array}{l} adapt : Unit \times Adapt \rightarrow Unit \\ trivAdapt : Adapt \\ merge : Adapt \times Adapt \nrightarrow Adapt \\ \hline \forall\, u : Unit \bullet adapt(u, trivAdapt) = u \\ \forall\, u : Unit;\ a_1, a_2 : Adapt \bullet \\ \qquad adapt(u, merge(a_1, a_2)) = adapt(adapt(u, a_1), a_2) \end{array}$$

We model *merge* as a partial function, reflecting the fact that it may not always be possible to merge two adaptations.

Finally a method (or methods if there are different kinds of units) for matching units is defined. The matching method must conform to the *matches* predicate below, which is defined in terms of the satisfies relationship and the adaptation function. For this paper it is sufficient to define abstractly what it means for two units to match.

A unit u_1 is said to match a second unit u_2 if and only if there are adaptations a_1 and a_2, such that u_1 adapted with respect to a_1 satisfies u_2 with respect to a_2.

This is captured by the predicate *matches*:

$$matches : \mathbb{P}(Unit \times Unit \times Adapt \times Adapt)$$

$$\forall u_1, u_2 : Unit;\ a_1, a_2 : Adapt \bullet$$
$$(u_1, u_2, a_1, a_2) \in matches \Leftrightarrow adapt(u_1, a_1)\ Satisfies\ adapt(u_2, a_2)$$

For generality we include an adaptation of both units. This differs from our earlier model [6], where only one unit (the library unit) was adapted. This generalised model reflects the fact that in some instances program units may also include adaptable elements, such as parameters. Such adaptable elements are most often introduced during the matching process where partial module adaptations are returned.

2.2 Module matching

The framework for matching units is extended to handle coarser-grained components, referred to here as *modules*. To maintain generality, we shall not formally model modules here; in particular no assumptions shall be made about the structuring of modules.

$$[Module]$$

We will assume that module interfaces differentiate between units that are *provided* by the module, and those that are *required* by the module. Required units are common in modules that define algorithmic or data refinements, together with modules that define adaptation schemes. To access the set of units in a module we assume that the functions *provided* and *required*, which return the set of units provided and required by the module, have been defined:

$$provided : Module \rightarrow \mathbb{F}\ Unit$$
$$required : Module \rightarrow \mathbb{F}\ Unit$$

The exact nature of these functions will depend on the particular modules in question. For a flat module, they will just return the set of units contained in the module. However for hierarchically structured modules they 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.

We model support for two kinds of module adaptation in this paper: adapting individual units; and module *subsetting*. Subsetting returns a submodule, itself a module, obeying the same syntactic and semantic constraints of its parent [6].

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.

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

We assume that there is a function *adapt* that applies a module adaptation to a module, returning a new module.

$$adapt : Module \times ModAdapt \rightarrow Module$$

3 Search tactics

To date existing retrieval tools based on specification matching have been restricted to performing a single matching step. In general terms, such retrieval tools take a collection of unit queries and return a set of candidate module matches (some approaches simply return the modules that match, while others also return the corresponding module adaptation).

We generalise this to a series of module matches, where the results from one match step form part of the input into the next matching step. To do this we define the notion of a *search tactic*, which generalises any combination of module matching steps. A search tactic is analogous to proof tactics used in interactive theorem provers [4]. Proof tactics are designed to combine multiple simple proof steps into a single more complicated step, thus automating the more mundane steps. Analogously, we aim to combine multiple individual search steps into a single complex search step.

We model a search tactic as a function that takes a list of specified-only program units (queries) and returns a set of solutions. In keeping with the terminology of interactive theorem provers, we shall refer to the list of specified-only program units as subgoals. Each solution consists of a sequence of module adaptations, a single local program adaptation and a new list of subgoals.

$$searchtac \mathrel{\widehat{=}} \text{seq}\, Unit \rightarrow \mathbb{F}(\text{seq}\, ModAdapt \times Adapt \times \text{seq}\, Unit)$$

A search tactic represents a set of solutions to an input query. Each solution includes the *search history*, represented by a sequence of module adaptations. Each solution also identifies any local adaptation that must be applied to the initial subgoals, as well as introducing a new list of subgoals.

We begin by defining two special tactics, *ID* and *FAIL*.

$$ID : searchtac$$
$$FAIL : searchtac$$

Applying the *ID* tactic always succeeds, returning a trivial result, which leaves the list of subgoals unchanged. The trivial result is represented by an empty list of module adaptations, ($\langle\rangle$), and a trivial local adaptation.

$$\forall x : \text{seq } Unit \bullet ID\ x = \{(\langle\rangle, trivAdapt, x)\}$$

In contrast, the *FAIL* tactic always fails, returning an empty set of results, ($\varnothing$), for any initial list of subgoals.

$$\forall x : \text{seq } Unit \bullet FAIL\ x = \varnothing$$

The first set of tactics that we describe are those that correspond to the simple one step tactics currently used in specification matching based retrieval tools [6]. Each of these tactics takes a library module as input to the matching process.

$$MATCHALL : Module \rightarrow searchtac$$
$$MATCHSOME : Module \rightarrow searchtac$$
$$MATCHONE : Module \rightarrow searchtac$$

The *MATCHALL* search tactic 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 as separate unit subgoals, and searching the library using the *MATCHALL* tactic, only modules that satisfy all of the requirements are returned.

For a module m, *MATCHALL* m defines a tactic which, given a list of subgoals, us, returns a set of triples of the form (mas, a, us'). For each triple:

- the sequence mas contains a single module adaptation ma, such that every unit in us (after they have been adapted with respect to a) matches a unit provided by the module m (adapted with respect to ma);
- the adaptation a corresponds to an adaptation of the units in the us;
- the list of units us' corresponds to the collection of unit subgoal contained within m after it has been adapted with respect to ma.

The *MATCHALL* tactic is specified as follows:

$$MATCHALL : Module \rightarrow searchtac$$

$$\begin{aligned}
&\forall m : Module;\ mas : \text{seq } ModAdapt;\ a : Adapt;\ us, us' : \text{seq } Unit \bullet \\
&\quad (mas, a, us') \in MATCHALL\ m\ us \Leftrightarrow \\
&\qquad us \neq \langle\rangle \wedge \exists ma : ModAdapt \bullet mas = \langle ma \rangle \wedge \\
&\qquad\quad \forall q : us \bullet \exists u : provided(unitsOf(m)) \bullet \\
&\qquad\qquad matches(u, q, \pi_1 ma, a) \wedge u \in \pi_2 ma \wedge \\
&\qquad\qquad\quad us' = required(unitsOf(adapt(m, ma)))
\end{aligned}$$

Note that the *MATCHALL* tactics fails (returns an empty answer set) when the subgoal list, us, is empty ($us = \langle\rangle$). This is necessary to ensure that recursive tactics will terminate once all unit specifications have been implemented.

The *MATCHSOME* tactic is a relaxation of the stricter *MATCHALL* tactic. This tactic succeeds when the specified module matches some (at least one) of the subgoals. This will include the set of matches formed by the *MATCHALL* tactic.

The *MATCHONE* tactic enables the user to include a number of alternate subgoals in order to find a single desired unit. This is useful 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 $s \neq \langle\rangle$. (Details are given in an earlier paper [6]; however like *MATCHALL*, we add the extra condition that these tactics fail when the unit specification list is empty).

The following tactics are useful when combining a number of search steps. Each search step may introduce new subgoals, which are appended to the end of the subgoal list. The tactics below give us a way of searching for matches for the oldest subgoals first, before looking for matches for newly added subgoals. In effect they enable breadth first searching strategies to be implemented.

$$MATCHFIRST : Module \rightarrow searchtac$$
$$MATCHFIRSTN : Module \rightarrow searchtac$$

The *MATCHFIRSTN* search tactic succeeds if there is some prefix of the subgoal list such that all units in this subsequence can be matched against units in the library module.

$$\begin{array}{|l} MATCHFIRSTN : Module \rightarrow searchtac \\ \hline \forall\, m : Module;\ mas : \operatorname{seq} ModAdapt;\ a : Adapt;\ us, us' : \operatorname{seq} Unit \bullet \\ \quad (mas, a, us') \in MATCHFIRSTN\ m\ us \Leftrightarrow \\ \qquad \exists\, us_1, us_2, us_3 \bullet us = us_1 \frown us_2 \wedge us_1 \neq \langle\rangle \wedge \\ \qquad\quad (mas, a, us_3) \in MATCHALL\ m\ us_1 \wedge us' = us_2 \frown us_3 \end{array}$$

We include the condition that the prefix sequence should be non-empty ($us_1 \neq \langle\rangle$). This ensures that the tactic fails if the initial subgoal list is empty.

The *MATCHFIRST* tactic can be defined as a special case of *MATCHFIRSTN*, with the prefix us_1 containing a single element equal to the head of the subgoal list us and the remaining units us_2 equal to the tail of the subgoal list.

4 Tacticals

To date we have redefined existing single step module matching techniques in a way that is consistent with the general search tactic form. In this section we show

how more complex search tactics can be constructed using tacticals. Tacticals are a mechanism that allow individual tactics to be combined. The tacticals defined in this section are based on those originally proposed for use in interactive theorem provers [4]. However their definition is quite different due to the fact that we not only return the final set of subgoals, but also the search history, at the completion of the search tactic.

The tacticals are summarised as follows, more details on each tactical are given below:

$$\begin{aligned} \mathit{THEN} &: \mathit{searchtac} \times \mathit{searchtac} \rightarrow \mathit{searchtac} \\ \mathit{ORELSE} &: \mathit{searchtac} \times \mathit{searchtac} \rightarrow \mathit{searchtac} \\ \mathit{APPEND} &: \mathit{searchtac} \times \mathit{searchtac} \rightarrow \mathit{searchtac} \\ \mathit{TRY} &: \mathit{searchtac} \rightarrow \mathit{searchtac} \\ \mathit{COMPLETE} &: \mathit{searchtac} \rightarrow \mathit{searchtac} \\ \mathit{REPEAT} &: \mathit{searchtac} \rightarrow \mathit{searchtac} \end{aligned}$$

The *THEN* tactical implements sequential composition of search steps. It combines two search steps, with the results of the first search step, if successful, being used as input into the second search step.

$$\begin{array}{|l} \mathit{THEN} : \mathit{searchtac} \times \mathit{searchtac} \rightarrow \mathit{searchtac} \\ \hline \forall\, t_1, t_2 : \mathit{searchtac};\ x : \operatorname{seq} \mathit{Unit};\ \mathit{mas} : \operatorname{seq} \mathit{ModAdapt}; \\ \quad a : \mathit{Adapt};\ \mathit{us} : \operatorname{seq} \mathit{Unit} \bullet (\mathit{mas}, a, \mathit{us}) \in (t_1\ \ \mathit{THEN}\ \ t_2)(x) \Leftrightarrow \\ \quad\quad \exists\, \mathit{mas}_1, \mathit{mas}_2 : \operatorname{seq} \mathit{ModAdapt};\ a_1, a_2 : \mathit{Adapt};\ \mathit{us}_1 : \operatorname{seq} \mathit{Unit} \bullet \\ \quad\quad\quad (\mathit{mas}_1, a_1, \mathit{us}_1) \in t_1(x) \land \\ \quad\quad\quad\quad ((\mathit{us}_1 = \langle\,\rangle \land \mathit{mas} = \mathit{mas}_1 \land a = a_1) \lor \\ \quad\quad\quad\quad ((\mathit{mas}_2, a_2, \mathit{us}) \in t_2(\mathit{us}_1) \land \mathit{mas} = \mathit{mas}_1 \frown \mathit{mas}_2 \land \\ \quad\quad\quad\quad\quad\quad (a_1, a_2) \in \operatorname{dom} \mathit{merge} \land a = \mathit{merge}(a_1, a_2))) \end{array}$$

This tactical calculates the results for the first tactic. If this tactic returns an empty list of subgoals (i.e., all specifications have been implemented), then the tactic stops, returning the answer from the first tactic. Otherwise, the subgoal list returned by the first tactic is passed as input to the second tactic. The resulting subgoal list in the solution is the list returned by the second tactic. The sequence of module adaptations returned by the *THEN* tactical corresponds to the module adaptation returned by the first tactic appended to the front of the sequence returned by the second tactic. The local adaptation returned corresponds to the merger of the local adaptations for the two tactics. These two local adaptations must be mergeable for a result to be returned.

The *ORELSE* tactical applies two alternative search tactics in sequence. The second tactic is only applied if the first tactic does not return any solutions. If the first search tactic is successful (returning a non-empty set of solutions), then the

tactical returns its results; otherwise the second search step is performed and its set of results is returned.

$$
\begin{array}{|l}
ORELSE : searchtac \times searchtac \rightarrow searchtac \\
\hline
\forall\, t_1, t_2 : searchtac;\ x : \mathrm{seq}\ Unit;\ mas : \mathrm{seq}\ ModAdapt; \\
\quad a : Adapt;\ us : \mathrm{seq}\ Unit \bullet (mas, a, us) \in (t_1\ ORELSE\ t_2)(x) \Leftrightarrow \\
\quad\quad (mas, a, us) \in t_1(x) \lor (t_1(x) = \varnothing \land (mas, a, us) \in t_2(x))
\end{array}
$$

The *APPEND* tactical is similar to the *ORELSE* tactic, except it performs a more exhaustive search. It will return the results of the second search step regardless of whether or not the first step was successful.

$$
\begin{array}{|l}
APPEND : searchtac \times searchtac \rightarrow searchtac \\
\hline
\forall\, t_1, t_2 : searchtac;\ x : \mathrm{seq}\ Unit;\ mas : \mathrm{seq}\ ModAdapt; \\
\quad a : Adapt;\ us : \mathrm{seq}\ Unit \bullet (mas, a, us) \in (t_1\ APPEND\ t_2)(x) \Leftrightarrow \\
\quad\quad (mas, a, us) \in t_1(x) \lor (mas, a, us) \in t_2(x)
\end{array}
$$

Using the *APPEND* tactical we generalise the basic tactics from the previous section, which are applied to a single module, to tactics that search over a list of library modules.

$$
\begin{aligned}
MATCHALL* &: \mathrm{seq}\ Module \rightarrow searchtac \\
MATCHSOME* &: \mathrm{seq}\ Module \rightarrow searchtac \\
MATCHONE* &: \mathrm{seq}\ Module \rightarrow searchtac \\
MATCHFIRST* &: \mathrm{seq}\ Module \rightarrow searchtac \\
MATCHFIRSTN* &: \mathrm{seq}\ Module \rightarrow searchtac
\end{aligned}
$$

For a list of modules $m_1, .., m_n$, *MATCHALL** is defined as:

$$
\begin{aligned}
&MATCHALL*(\langle m_1, .., m_n \rangle) \mathrel{\widehat{=}} \\
&\qquad MATCHALL(m_1)\ APPEND\ ..\ APPEND\ MATCHALL(m_n)
\end{aligned}
$$

The other generalised tactics above are defined in a similar manner.

The *TRY* tactical will attempt to apply a tactic, but if it fails then it will leave the list of subgoals unchanged.

$$\forall\, t : searchtac \bullet TRY\ t = t\ ORELSE\ ID$$

The *COMPLETE* tactical only succeeds when the tactic it is applying matches all of the initial subgoals and does not return any new subgoals.

$$
\begin{array}{|l}
COMPLETE : searchtac \rightarrow searchtac \\
\hline
\forall\, t : searchtac;\ x : \mathrm{seq}\ Unit \bullet (mas, a, us) \in COMPLETE\ t\ x \Leftrightarrow \\
\quad (mas, a, us) \in t\ x \land us = \langle\,\rangle
\end{array}
$$

The *REPEAT* tactical enables us to apply a tactic repeatedly, at each stage applying the tactic to the result of the previous application. It will only return results from the last repetition.

$$\forall\, t : searchtac \bullet REPEAT\ t = (t\ \ THEN\ \ REPEAT\ t)\ \ ORELSE\ \ ID$$

The *REPEAT* tactical is defined recursively. It repeatedly applies t to the current list of subgoals, terminating either when the tactic t fails, or when t returns an empty subgoal list.

5 Example

As a proof of concept, the tactics and tacticals have been implemented as an extension to a retrieval tool already used to support the CARE method [7]. The tactics have been applied to several simple examples, including one for summing the lengths of two lists, which we will describe in this section.

Fig. 1 contains a CARE program for summing the length of two lists. The main fragment ***sumlengths*** takes two lists as input, and returns a natural number output that is equal to the sum of the lengths of the two lists. It is implemented by applying the fragment ***length*** to the two input lists, then applying the fragment ***add*** to the results. The program includes specifications for the fragments ***length*** and ***add***, together with type declarations for lists (of natural numbers) and natural numbers.

type *Nat* == $\mathbb{N}$
type *List* == $\mathrm{seq}\,\mathbb{N}$

sum_lengths(**in** *x,y:List,* **out** *z:Nat*)
  post $z = len(x) + len(y)$
::= length(x)::u:Nat; length(y)::v:Nat; add(u,v)::z:Nat;z.

length(**in** *x:List,* **out** *n:Nat*)
  post $n = len(x)$.
add(**in** *x,y:Nat,* **out** *z:Nat*)
  post $z = x + y$.

Fig. 1. A program for summing the lengths of two lists

In an earlier paper [5] we show how ***sumlengths*** can be implemented using only library components. This is done by combining primitives from the two library modules shown in Fig. 2.

The NAT module (partial listing shown in Fig. 2(a)), defines primitives for representing and manipulating natural numbers. The module includes operations for adding and multiplying two natural numbers, together with fragments for returning the constants zero and one. The operators are specified using standard arithmetic operators.

```
module NAT is
  type Nat== ℕ
  add(in x,y:Nat, out z:Nat)
    post z = x + y.
  mult(in x,y:Nat, out z:Nat)
    post z = x * y.
  zero(out z:Nat)
    post z = 0.
  one(out z:Nat)
    post z = 1.
end module.
```

(a) Natural number module

```
module LIST[X] is
  type List== seq X.
  nil(out y:X)
    post y = ⟨⟩.
  head(in x:List, out y:X)
    pre x ≠ ⟨⟩
    post y = head x.
  tail(in x:List, out y:List)
    pre x ≠ ⟨⟩
    post y = tail x.
  cons(in x:X,in y:List,out z:List)
    post z = ⟨x⟩ ⌢ y.
  append(in x,y:List, out z:List)
    post z = x ⌢ y.
  length(in x:List, out n:Nat)
    post n = len(x).
end module.
```

(b) List module

Fig. 2. CARE modules

The LIST module (partial listing shown in Fig. 2(b)) provides primitives for creating and manipulating lists. The module is parameterised over the set of values (X) that individual list elements can take. Primitive operations are provided for: creating an empty list; accessing the head of a list; accessing the tail of a list; adding an element to the head of a list; joining two lists; and calculating the length of a list.

Two adaptation templates are also required for this development. The first of these, the template FUNDECOMP, shown Fig. 3, lets us solve a problem by breaking it down into subproblems that can be solved sequentially. The ***main*** fragment computes an answer by applying functions g and h to the input arguments, then joining the results by applying a third function f. The template is parameterised over these three functions, so it can be adapted to solve a variety of problems.

The second library template, DROPINPUT, shown in Fig. 4 allows a fragment (***main***) to be implemented by calling a secondary fragment (***frag1***) with one less input argument. This template can be applied when the input argument to be dropped does not occur in the main fragment. This is useful because specification matching only matches against fragments with exactly the same number of arguments, even if the preconditions and postconditions are the same.

The (manual) library based development of ***sumlengths*** begins by matching ***sumlengths*** with the ***main*** fragment from FUNDECOMP (see Fig. 3) with the fol-

```
template FunDecomp[X; Y; U; V; W; f : U * V → W; g : X * Y → U;
    h : X * Y → V] is
  main(in x:X, in y:Y, out w:W)
    pre P(x, y)
    post w = f(g(x, y), h(x, y))
  ::= gfrag(x,y)::u:U;
      hfrag(x,y)::v:V;
      ffrag(u,v).
  ffrag(in u:U,in v:V, out w:W)
    post w = f(u, v).
  gfrag(in x:X, in y:Y, out u:U)
    pre P(x, y)
    post u = g(x, y).
  hfrag(in x:X,in y:Y, out v:V)
    pre P(x, y)
    post v = h(x, y).
end template.
```

Fig. 3. A sequential decomposition template

```
template DropInput[X; Y; Z; P : X → 𝔹; f : X → Z] is
  fragment main(in x:X,in y:Y,out z:Z)
    pre P(x)
    post z = f(x)
  ::= frag1(x).
  fragment frag1(in x:X, out z:Z)
    pre P(x)
    post z = f(x).
end template.
```

Fig. 4. A wrapper template for dropping unused inputs

lowing parameter instantiations:

$$f \mapsto \lambda\, u, v \bullet u + v, g \mapsto \lambda\, u, v \bullet \mathit{len}(u)$$
$$h \mapsto \lambda\, u, v \bullet \mathit{len}(v), P \mapsto \lambda\, u, v \bullet \text{true}$$

After applying these instantiations to the template we get the following subgoals (we omit the trivial preconditions):

```
ffrag(in u:Nat,in v:Nat,out w:Nat)
  post w = u + v.
gfrag(in x:List,in y:List,out u:Nat)
  post u = len(x).
hfrag(in x:List,in y:List,out v:Nat)
```

post $v = len(y)$.

The development continues by finding implementations for these three fragments. The first of these fragments can be matched against the fragment `add` from the NAT module using exact matching. However the other two fragments cannot yet be matched directly against module fragments. Despite their postcondition being the same as that of the `length` fragment from the LIST module, `gfrag` and `hfrag` have an extra (unused) argument. These fragments can be adapted by applying the DROPINPUT template. For `gfrag`, the DROPINPUT template can be applied by instantiating the parameters as follows:

$$P \mapsto \text{true}, f \mapsto \lambda\, u \bullet len(u)$$

The following specified-only fragment is returned:

```
frag1(in x:List, out u:Nat)
```
post $u = len(x)$.

The fragment `frag1` can be matched against the fragment `length` from the LIST template. To complete the example the fragment `hfrag` is matched in a similar manner.

The above development can be automated by applying the following tactic:

$$\begin{array}{c} COMPLETE(MATCHFIRSTN(FunDecomp)\ THEN \\ REPEAT\ (MATCHFIRST * (\langle Nat, List\rangle))\ ORELSE \\ MATCHFIRST(DropInput))) \end{array}$$

We use the *COMPLETE* tactical so that only total matches are returned, i.e., where subgoals are returned. This is applied to a subtactic which uses the *THEN* tactical. The first part of this tactic calls the sequential decomposition template to break the problem into smaller parts. Then we repeatedly try to match the first of the resulting subgoals against a unit from the data structure modules; or if this fails then drop an input from the first subgoal in the list. In general this tactic terminates when the tactic inside of the *REPEAT* tactical fails. This happens either when there are no remaining subgoals (i.e., all requirements have been met), or there is a unit at the head of the subgoal list that does not match a unit from the data structure modules, and cannot have any input arguments removed.

6 Related work

There are a number of approaches to component retrieval based on formal specification matching [12,8,15]. These approaches all fit within our general framework. There has been less focus on the problem of adaptation of formally specified components. However the approaches of Penix and Alexander [10], and the SPARTACAS project [9], are similar to the CARE approach to component adaptation. The main

difference is the use of branching fragments in our approach, used to define adaptation templates that combine components by doing case analysis on the inputs [5]. In our approach branching conditions are defined explicitly and can often be deduced through the matching process. This differs to the related approaches, where the branching condition is implicit and more often requires human interaction to deduce.

Bracciali [2] describes an approach to adapting mismatching behaviours. Their methodology has three main features: component interfaces; adaptor specifications; and adaptor derivation. Components interfaces include a specification of the functions offered and required (via signatures), together with the specification of the behaviour (the interaction protocol). Adaptor specifications specify interoperation between two components. Adaptor derivation automatically generates a concrete adaptor. Adaptor derivation is based on matching function signatures; in this sense our approach is more sophisticated. However, their approach takes into account non-functional behaviour and they use π-calculus to model more complex connectors (architectures).

Bosch [1] proposes a slightly different approach to those discussed already. Rather than defining general adaptation techniques that can be applied to any component, Bosch instead associates adaptation techniques with particular components. Such an approach allows Bosch to not only define adaptations similar to the ones discussed in this paper (so called black-box adaptations), but also white-box adaptations that require knowledge of the internals of a component. However it is not clear that such an approach could be integrated with formal-based retrieval.

7 Conclusions

In this paper we have defined a number of general search tactics that can be used to find a solution to a problem using a library of formally specified components. We have defined basic tactics, which allow us to find direct matches, together with tacticals, allowing us to combine matching steps and in effect enable us to apply a combination of adaptation steps. The tactics are general and should be applicable to a variety of formal CBSD approaches. By defining tactics we can eliminate many of the tedious adaptation steps, allowing the user to concentrate purely on the creative steps in the development. Like interactive theorem proving, the use of search tactics allows the user to have as little or as much interaction during the development as they desire, or as is required. The work presented in this paper represents a significant advance on previous work. The tactics have been prototyped by extending the CARE tools and tested on several small examples.

Acknowledgements

This work was funded by Australian Research Council Discovery Grant DP0208046, Compilation of Specifications. Thanks to Colin Fidge, who provided useful feedback on an earlier draft of this paper.

References

[1] Bosch, J., *Superimposition : A component adaptation technique*, Information and Software Technology **4** (1999), pp. 249–305.

[2] Bracciali, A., A. Bragi and C. Canal, *Adapting components with mismatching behaviours*, in: J. Bishop, editor, *Proceedings of CD'2002*, LNCS **2370** (2002), pp. 185–199.

[3] Brereton, P. and D. Budgen, *Component-based systems: A classification of issues*, IEEE Computer **33** (2000), pp. 54–62.

[4] Gordon, M. J., A. J. Milner and C. P. Wadsworth, "Edinburgh LCF : A Mechanised Logic of Computation," Lecture Notes in Computer Science **78**, Springer-Verlag, 1979.

[5] Hemer, D., *A formal approach to component adaptation and composition*, in: V. Estivill-Castro, editor, *Proceedings of ACSC'2005*, CRPIT **38** (2005), pp. 259–266.

[6] Hemer, D. and P. Lindsay, *Specification-based retrieval strategies for module reuse*, in: D. Grant and L. Stirling, editors, *Proc. of Australian Software Engineering Conference (ASWEC'2001)* (2001), pp. 235–243.

[7] Hemer, D. and P. Lindsay, *Template-based construction of verified software*, IEE Proceedings Software Engineering **152** (2005), pp. 2–14, special issue on Reusable Software Libraries.

[8] Jeng, J.-J. and B. Cheng, *Specification matching for software reuse: A foundation*, in: *Proceedings of ACM Symposium on Software Reuse*, 1995, pp. 97–105.

[9] Morel, B. and P. Alexander, *SPARTACAS: Automating component reuse and adaptation*, IEEE Transactions on Software Engineering **30** (2004), pp. 587–600.

[10] Penix, J. and P. Alexander, *Toward automated component adaptation*, in: *Proceedings of the Ninth International Conference on Software Engineering and Knowledge Engineering*, 1997, pp. 535–542.

[11] Penix, J. and P. Alexander, *Efficient specification-based component retrieval*, Automated Software Engineering **6** (1999), pp. 139–170.

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

[13] Prieto-Diaz, R. and F. P, *Classifying software for reusability*, IEEE Software **4** (1987), pp. 6–16.

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

[15] Zaremski, A. M. and J. Wing, *Specification matching of software components*, ACM Transactions on Software Engineering **6** (1997), pp. 333–369.