A Formal Approach to Component Adaptation and Composition

David Hemer

School of Computer Science,
The University of Adelaide, SA, 5005, Australia
Email: hemer@cs.adelaide.edu.au

Abstract

Component based software engineering (CBSE), can in principle lead to savings in the time and cost of software development, by encouraging software reuse. However the reality is that CBSE has not been widely adopted. From a technical perspective, the reason is largely due to the difficulty of locating suitable components in the library and adapting these components to meet the specific needs of the user.

Formal approaches to retrieval – using formal notations for interface specification, and semantic based matching techniques – have been proposed as a solution to the retrieval problem. These approaches are aimed at overcoming the lack of precision and ambiguity associated with text-based component interfaces, requirements and retrieval techniques. However these approaches fail to adequately address the problem of component adaptation and composition.

In this paper we describe how component adaptation and composition strategies can be defined using parameterised library templates. We define a variety of templates, including wrapper templates that adapt a single program component, and architecture templates that combine program components. We include definitions for sequential architectures, independent architectures and alternative architectures. These library templates are formally specified, so we are able to employ existing formal-based retrieval strategies to match problem specifications against library templates. We discuss how adaptation and composition can be semi-automated by the library templates defined in this paper in combination with existing retrieval strategies.

Keywords: Component-based Software Engineering, Component Adaptation, Retrieval, Specification matching

1 Introduction

Component-based software engineering (CBSE) is an approach to constructing software programs using a library of components. The idea is to build programs by composing various library components, rather than building the program from scratch. Since CBSE encourages software reuse, there are potential savings on development time and costs.

While the concept was first introduced by McIlroy (1969) more than thirty years ago, CBSE has at best been slowly adopted. Moreover in safety critical applications, where the use of CBSE could lead to savings in verification effort, uptake of CBSE has been minimal. There are both technical and non-technical reasons for this. From the technical perspective, two of the main challenges (besides from actually populating the library), are finding components that satisfy the needs of the software engineer (*retrieval*), and adapting library components to fit the software engineers specific requirements (*adaptation* and *composition*).

Traditional keyword-based retrieval strategies have inherent problems, such as the ambiguity and imprecision associated with informal specifications. To overcome these problems, formal approaches to retrieval, referred to as *specification matching*, have been proposed (Zaremski & Wing 1997, Perry & Popovich 1993, Jeng & Cheng 1995, Schumann & Fischer 1997). Specification matching strategies compare component interfaces and problem specifications (or user requirements) that have both been formally specified. Matching of formal specifications involves establishing that a logical equivalence or relationship holds between the specifications. Tool support is usually based on theorem prover technology.

While these approaches show some promise, they largely ignore the related problems of component adaptation and composition. So while these matching techniques can be used to locate components that partially satisfy the user requirements, they provide no insight into how they can be adapted to fully satisfy the user requirements.

In this paper we propose using *templates* from the CARE language (Hemer & Lindsay 2004, Lindsay & Hemer 1997) to define adaptation strategies for modifying and combining components. Templates are modular collections of basic CARE units. Templates can be parameterised over their functional behaviour by including higher-order parameters. The use of parameters lets us define very general adaptation strategies that can be applied to a variety of problems. We discuss briefly how specification matching strategies can be used to semi-automate the adaptation and composition process.

In Section 2 we give a brief overview of the CARE language, focusing on those parts of the language used in this paper. In this section we introduce CARE modules and CARE templates. Sections 3-6 define a variety of new templates that are used to adapt and compose CARE components. We define wrapper templates (Section 3), which are used to adapt single components; sequential architectures (Section 4), used to solve a problem by sequentially combining components; independent architectures (Section 5), used to solve a problem by breaking it up into subproblems that are solved independently; and alternative architectures (Section 6), used to solve a problem by case analysis. We do not discuss iterator architectures in this paper; however a generic accumulator template is defined in a related paper by Hemer &

This work was completed within the School of Information Technology and Electrical Engineering, The University of Queensland.

Lindsay (2004). In Section 7 we discuss how specification matching techniques can be used to semi-automatically apply adaptation templates. In Section 8 we summarise related work.

2 The CARE formal development environment

The adaptation strategies described in this paper are defined in the CARE language (Lindsay & Hemer 1996, Lindsay & Hemer 1997). While most of the ideas are more widely applicable, there are certain features of the CARE language that make it more suitable for defining adaptation templates than other languages. We discuss these differences briefly in Section 7.

CARE is a language, methodology and set of tools designed to generate verified code from high-level specifications. The language is functional in nature, with a program consisting of units such as types and fragments. Types model data structures; fragments are similar to the functions used in functional programming languages (e.g. ML and Miranda). Each unit is formally specified using a Z-like specification language (Spivey 1989). Units can either be implemented by calls to other units, or by calls to primitives from the target code language. A development is complete when all types and fragments have been implemented. At this point target language code can be automatically generated, and proof obligations that establish the correctness of the generated code with respect to the initial specification can be generated and discharged.

CARE supports the notion of reuse by providing a library of *modules* and *templates*. Both modules and templates consist of a collection of units. We begin by briefly describing the main units of the language: types and fragments.

2.1 The CARE language

Types are declared by giving an identifier and specifying the set of values that objects of this type may take. For example a type `NatList` representing all lists of natural numbers can be declared as:

type `NatList ==` $seq\,\mathbb{N}$.

where $\mathrm{seq}\,\mathbb{N}$ represents the set of all sequences of natural numbers. The specification provides an abstract representation of the type; a concrete representation is provided by the implementation. Types can be implemented either by calls to primitives from the target language, or by combining other types (Lindsay & Hemer 1997). An abstract (specified) type may have more than one possible concrete implementation. For example the natural number abstract type could be implemented by 16 or 32 bit, signed integers or unsigned integers.

Fragments are either simple or branching. Simple fragments, which return a result based on the value of the inputs, are declared by giving an identifier, a list of input and output variables, an optional precondition and a postcondition. The precondition and postcondition are used to generate proof obligations that establish the correctness of fragments. For example the fragment `divide` for dividing natural numbers, provided the denominator is non-zero, can be specified as:

fragment `divide(`**in** `n,d:Nat,`**out** `r:Nat)`
  pre $d \neq 0$
  post $r = n \; div \; d$.

Branching fragments are used to control flow (they describe what branch should be taken depending on the values of the inputs), but may also return outputs at any of the branches. In their most general form, a branching fragment is declared by giving an identifier, a list of input variables, an optional precondition, and two or more result branches. Each result branch includes a guard, defining the set of inputs for which this branch will be used, together with an optional list of output variables and an optional postcondition. The guard for the final branch may be omitted, in which case the guard is just the negation of all other guards. In this paper we will only consider branching fragments that control flow but do not return any outputs and have exactly two branches. Such branching fragments can be specified by providing a predicate (the *test*) on the inputs; the guard for the first branch is the test, while the guard for the second branch is the negation of the test. An example is the fragment `isempty`, which tests whether a list is empty.

bfragment `isempty(`**in** `s:List)`
  test $s = \langle\,\rangle$.

The test $s = \langle\,\rangle$ represents the guard for the first branch. The guard for the second branch is the negation of this test, i.e., $\neg\, s = \langle\,\rangle$.

Fragments can be implemented either by calling primitives from the target language directly or by calling other fragments. We will not go into any details about the first category of implementations, except to say that we generally prohibit the user from defining such implementations, instead restricting such implementations to predefined library components. In the latter case a fragment implementation may include: calls to simple fragments; bindings to local variables; calls to branching fragments for control flow; and sequencing. For example, consider the following recursive implementation of a fragment for calculating the sum of a list of naturals numbers:

fragment `sum(`**in** `s:NatList,` **out** `n:Nat)`
  post $n = \sum_{i=1}^{len(s)} s(i)$
`::=` **if** `isempty(s)`
    then `zero`
    else `head(s)::m:Nat;`
      `tail(s)::t:List;`
      `add(m,sum(t))`

In this example it is assumed the fragment `zero` returns the constant 0, `add` adds two natural numbers, `head` returns the first element of a non-empty list and `tail` returns the remainder of a non-empty list after the first element has been removed. The call to `isempty` is used to control flow, with the first (terminating) branch taken if the list is empty, and second branch taken otherwise.

2.2 Modules

Modules are collections of related fragments and types. A common usage of modules in CARE is for defining data structures, together with operations on the data structure. Modules can be parameterised over sets of values, meaning that polymorphic data structures can be defined. To use a parameterised module, any set parameters must be instantiated to an actual set. We give two examples of data structure modules, one for representing natural numbers, the other for representing lists of elements. The second module is an example of a polymorphic data structure, with the module parameterised over the type of the individual list elements.

The `Nat` module (only a partial listing for space reasons), shown below defines primitives for representing and manipulating natural numbers. The implementation details have been left hidden, but one

possible implementation in C would be to represent the underlying data structure as an unsigned integer.

module NAT **is**
  type *Nat* == $\mathbb{N}$
  fragment *add*(**in** *x,y:Nat*, **out** *z:Nat*)
    post $z = x + y$.
  fragment *mult*(**in** *x,y:Nat*, **out** *z:Nat*)
    post $z = x * y$.
  fragment *zero*(**out** *z:Nat*)
    post $z = 0$.
  fragment *one*(**out** *z:Nat*)
    post $z = 1$.
end module.

The module includes operations for adding and multiplying two natural numbers. The operators are specified using standard arithmetic operators. The NAT module also includes fragments *zero* and *one* for representing the constants 0 and 1.

The LIST module (only a partial listing for space reasons) 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. It also includes a branching fragment for testing whether an element is a member of a list. The function *elems* returns the set of all elements in a list (ignoring repetitions), and the function # adds a single element to the head of a list.

module LIST*[X]* **is**
type *List*== $seq\, X$.
fragment *nil*(**out** *y:X*)
  post $y = \langle\rangle$.
fragment *head*(**in** *x:List*, **out** *y:X*)
  pre $x \neq \langle\rangle$
  post $y = head\, x$.
fragment *tail*(**in** *x:List*, **out** *y:List*)
  pre $x \neq \langle\rangle$
  post $y = tail\, x$.
fragment *cons*(**in** *x:X*,**in** *y:List*,**out** *z:List*)
  post $z = x \# y$.
fragment *append*(**in** *x,y:List*, **out** *z:List*)
  post $z = x \frown y$.
fragment *length*(**in** *x:List*, **out** *n:Nat*)
  post $n = len(x)$.
bfragment *isin*(**in** *x:X*, **in** *y:List*)
  test $x \in elems(y)$.
end module.

2.3 Templates

The CARE language also includes generic library components, called *templates*. Like modules, templates consist of a collection of CARE units (types and fragments). However templates can be parameterised over functional behaviour, as well as over types. This is achieved by including higher-order parameters, corresponding to functions, relations and sets. To use a template these parameters must be instantiated to an actual value (in this case either a function, relation or set). We describe templates with a simple example. More template examples will be introduced later in the paper.

An example is the ADDCONSTANTARG template, shown below, which is similar to the "bind1st" template from the C++ Standard Template Library (Breymann 2000). This templates enables us to pass a constant appearing in the main fragment as an argument in a call to a secondary fragment. This allows us to solve a specific problem (e.g., adding 1 to a number) by using a more general fragment (e.g., adding two numbers).

We begin by defining the syntax and semantics of this template. The template implements the provided fragment *main*, which includes a single input argument, by a call to the fragment *gfrag*, which includes two input arguments. This is done by passing the input argument *x* of *main* as one argument of *gfrag*, and a constant value *afrag* as the other argument. A third fragment, *afrag*, representing the constant value is also required. The template is parameterised over the sets X, Y and Z; a constant value $a : Y$; and a predicate $Q : Y \times X \times Z \rightarrow \mathbb{B}$ (representing the post-condition of *main*). We assume that the template also includes types *X*, *Y* and *Z* representing the sets X, Y and Z respectively, however the declaration of these types has been dropped for brevity.

template ADDCONSTANTARG*[*X; Y; Z;
  $Q : Y \times X \times Z \rightarrow \mathbb{B}$; $a : Y$*]* **is**
fragment *main*(**in** *x:X*, **out** *z:Z*)
  post $Q(a, x, z)$
::= gfrag(afrag,x).
fragment *gfrag*(**in** *y:Y*, **in** *x:X*, **out** *z:Z*)
  post $Q(y, x, z)$.
fragment *afrag*(**out** *y:Y*)
  post $y = a$.
end template.

To illustrate how the ADDCONSTANTARG template is applied consider the following user requirement, which specifies a fragment for incrementing a natural number.

type *N* == $\mathbb{N}$.
fragment *increment*(**in** *m:N*, **out** *n:N*)
  post $n = m + 1$.

We assume that there are no fragments in the library (and in particular in the NAT module) that can implement our requirement directly. We can however apply the ADDCONSTANTARG template, replacing the *main* fragment by *increment*. To do this we adapt the template by instantiating the parameters as:

$$X, Y, Z \mapsto \mathbb{N}$$
$$Q \mapsto \lambda\, a, b, c \bullet c = b + a$$
$$a \mapsto 1$$

and renaming the local variables of *main* as $x \mapsto m, z \mapsto n$. Parameter instantiations are defined using lambda abstractions. The instantiation of Q above, will replace an expression of the form $Q(x, y, z)$ with $z = y + x$. This generates the following specified-only fragments:

fragment *gfrag*(**in** *y:N*, **in** *x:N*, **out** *z:N*)
  post $z = x + y$.
fragment *afrag*(**out** *y:N*)
  post $y = 1$.

The fragments *gfrag* and *afrag* can now be implemented directly by the fragments *add* and *one* respectively from the library module NAT, thus completing the example.

This completes the background to CARE. In the following sections we present a variety of new adaptation templates.

3 Wrapper templates

The first class of adaptation templates that we define are wrapper templates. As opposed to the other templates introduced later in the paper, these templates

do not combine fragments. Instead they take a single fragment and do some sort of modification on this fragment to provide a new fragment. We give two examples of wrapper templates in this section: the first modifies the input arguments; the second modifies the precondition.

The DROPINPUT template 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.

template DROPINPUT[X; Y; Z; $P : X \to \mathbb{B}$; $f : X \to Z$] **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.

We will illustrate the use of this template in Section 4.

The template WEAKENPRECOND allows a fragment (*frag*) to be implemented by another fragment (*frag1*) with a weaker precondition. This is useful, because in implementing a fragment f by another fragment g, we only have to ensure that the precondition of f implies the precondition of g. This template can be applied to a fragment whose precondition is the conjunction of two predicates (represented by the parameters P and Q). The precondition is weakened by dropping the second of the conjuncts. We provide other templates for dropping the first conjunct and dropping the entire precondition.

template WEAKENPRECOND[X; Y; $P : X \to \mathbb{B}$; $Q : X \to \mathbb{B}$; $R : X \times Y \to \mathbb{B}$] **is**
fragment *frag*(**in** x:X, **out** y:Y)
  pre $P(x) \wedge Q(x)$
  post $R(x, y)$
::= *frag1*(x).
fragment *frag1*(**in** x:X, **out** y:Y)
  pre $P(x)$
  post $R(x, y)$.
end template.

A similar template can be defined that weakens a precondition by adding disjuncts.

4 Sequential architectures

Sequential architectures allow problems to be solved by combining fragments sequentially. We give two examples in this section. The first example, FUNDECOMP, is a syntactic based adaptation strategy. The second, SEQDECOMP, is a semantic based adaptation strategy, which uses applicability conditions to place restrictions on parameter values. The FUNDECOMP template allows a problem to be functionally decomposed into subproblems that can be solved separately. 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.

template FUNDECOMP[X; Y; U; V; W; $f : U \times V \to W$; $g : X \times Y \to U$; $h : X \times Y \to V$] **is**
fragment *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).
fragment *ffrag*(**in** u:U, **in** v:V, **out** w:W)
  post $w = f(u, v)$.
fragment *gfrag*(**in** x:X, **in** y:Y, **out** u:U)
  pre $P(x, y)$
  post $u = g(x, y)$.
fragment *hfrag*(**in** x:X, **in** y:Y, **out** v:V)
  pre $P(x, y)$
  post $v = h(x, y)$.
end template.

To illustrate the use of the FUNDECOMP template consider the following specification for computing the sum of the lengths of two lists.

fragment *sum_lengths*(**in** x,y:L, **out** z:N)
  post $z = len(x) + len(y)$.

The fragment *sum_lengths* can be plugged into the FUNDECOMP template, replacing the *main* fragment. The template is adapted by instantiating the parameters as follows:

$$\begin{aligned} X, Y &\mapsto \text{seq}\,\mathbb{N} \\ U, V, Z &\mapsto \mathbb{N} \\ f &\mapsto \lambda\, a, b \bullet a + b \\ g &\mapsto \lambda\, a, b \bullet len(a) \\ h &\mapsto \lambda\, a, b \bullet len(b) \\ P &\mapsto \lambda\, a, b \bullet \text{true} \end{aligned}$$

After applying these instantiations to the template we get the following specified-only fragments that are added to the user program (we omit the trivial preconditions):

fragment *ffrag*(**in** u:N, **in** v:N, **out** w:N)
  post $w = u + v$.
fragment *gfrag*(**in** x:L, **in** y:L, **out** u:N)
  post $u = len(x)$.
fragment *hfrag*(**in** x:L, **in** y:L, **out** v:N)
  post $v = len(y)$.

The development continues by finding implementations for these three fragments. The first of these fragments can implemented directly by the fragment *add* from the NAT module. However the other two fragments cannot be implemented directly by module fragments yet, despite their postcondition being the same as that of the *length* fragment from the LIST module. This is because *gfrag* and *hfrag* have an extra (unused) argument. These fragments can be adapted by applying the DROPINPUT template. For *gfrag*, the 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:

fragment *frag1*(**in** x:L, **out** u:N)
  post $u = len(x)$.

The fragment *frag1* can now be implemented by the fragment *length* from the LIST template. To complete the example the fragment *hfrag* is implemented in a similar manner.

The second sequential architecture that we describe here is the SEQDECOMP template. Like the FUNDECOMP template, it allows us to solve a problem

by sequentially combining two other fragments. However it differs in that it is not purely syntax based. Instead the template includes three applicability conditions that place restrictions on the values that can be assigned to the parameters. To use the template these applicability conditions must be proven for the instantiated parameter values. The first condition states that the precondition of ***gfrag*** follows from the precondition of ***main***. The second condition states that the precondition of ***hfrag*** follows from the precondition of ***main*** and the postcondition of ***gfrag***. The third condition states that the postcondition of ***main*** holds, assuming the preconditions and postconditions of ***gfrag*** and ***hfrag***.

template SEQDECOMP`[X;Y;Z;` $P : X \to \mathbb{B}$`;` $Q : X \times Y \to \mathbb{B}$`;` $R : X \times Z \to \mathbb{B}$`;` $S : Z \to \mathbb{B}$`;` $U : X \to \mathbb{B}$; $V : Z \to \mathbb{B}$; $T : Z \times Y \to \mathbb{B}$`]` **is**
applic conds
$\forall x : X \bullet P(x) \Rightarrow U(x)$
$\forall x : X, z : Z \bullet P(x) \wedge R(x,z) \wedge S(z) \Rightarrow V(z)$
$\forall x : X, y : Y, z : Z \bullet U(x) \wedge R(x,z) \wedge S(z) \wedge V(z) \wedge T(z,y) \Rightarrow Q(x,y)$
fragment ***main*****(in** `x:X,` **out** `y:Y)`
pre $P(x)$
post $Q(x,y)$
`::= gfrag(x)::z:Z;`
`hfrag(z).`
fragment ***gfrag*****(in** `x:X,` **out** `z:Z)`
pre $U(x)$
post $R(x,z) \wedge S(z)$
fragment ***hfrag*****(in** `z:Z,` **out** `y:Y)`
pre $V(z)$
post $T(z,y)$.
end template.

To illustrate the use of the SEQDECOMP template, suppose we want to implement the following fragment for returning the maximum value in a non-empty list of natural numbers:

fragment ***max*****(in** `x:L,` **out** `m:N)`
pre $x \neq \langle\rangle$
post $m \in elems(x) \wedge \forall e \in elems(x) \bullet m \geq e$.

The strategy we employ to implement ***max***, is to use the SEQDECOMP template to combine the following two fragments, the first used to sort the list in ascending order, and the second to return the last element of a non-empty list (this is a very inefficient way to solve the problem and is used for illustration purposes only):

fragment ***sort*****(in** `x:L,` **out** `y:L)`
post $items(x) = items(y) \wedge ordered(y)$.
fragment ***last*****(in** `x:L,` **out** `m:N)`
pre $x \neq \langle\rangle$
post $m = last(x)$.

We plug these three fragments into the template, replacing ***main*** with ***max***, ***gfrag*** with ***sort***, and ***hfrag*** with ***last***. The template parameters are instantiated as follows:

$$
\begin{aligned}
X &\mapsto \mathrm{seq}\,\mathbb{N}\\
Y &\mapsto \mathbb{N}\\
Z &\mapsto \mathrm{seq}\,\mathbb{N}\\
P &\mapsto \lambda\, a \bullet a \neq \langle\rangle\\
Q &\mapsto \lambda\, a, b \bullet b \in elems(a) \wedge \forall e \in elems(a) \bullet b \geq e\\
U &\mapsto \lambda\, a \bullet \mathrm{true}\\
R &\mapsto \lambda\, a, b \bullet items(a) = items(b)\\
S &\mapsto \lambda\, a \bullet ordered(a)\\
V &\mapsto \lambda\, a \bullet a \neq \langle\rangle\\
T &\mapsto \lambda\, a, b \bullet b = last(a)
\end{aligned}
$$

After instantiation the applicability conditions become:

$$\forall x : \mathrm{seq}\,\mathbb{N} \bullet x \neq \langle\rangle \Rightarrow \mathrm{true}$$

$$\forall x : \mathrm{seq}\,\mathbb{N}, z : \mathrm{seq}\,\mathbb{N} \bullet x \neq \langle\rangle \wedge items(x) = items(z) \Rightarrow z \neq \langle\rangle$$

and

$$\forall x : \mathrm{seq}\,\mathbb{N}, y : \mathbb{N}, z : \mathrm{seq}\,\mathbb{N} \bullet y = last(z) \wedge ordered(z) \wedge items(x) = items(z) \Rightarrow y \in elems(x) \wedge \forall e \in elems(x) \bullet y \geq e$$

All of these conditions can be proven to be true.

5 Independent architectures

Independent architectures allow problems to be solved by splitting the problem into subproblems that can be solved independently, such that each subproblem does not rely on input or output from the other subproblems. An example is the template PARDECOMP, which implements the fragment ***main***, which returns two results, by calling two separate fragments that each return one of the results. Each subfragment only relies on the initial input value of the ***main*** fragment.

template INDEPDECOMP`[X;Y;Z;P :` $X \to \mathbb{B}$`;` $Q : X \times Y \to \mathbb{B}$`;` $R : X \times Z \to \mathbb{B}$`]` **is**
fragment ***main*****(in** `x:X,` **out** `y:Y,` **out** `z:Z)`
pre $P(x)$
post $Q(x,y) \wedge R(x,z)$
`::= qfrag(x)::y:Y;`
`rfrag(x)::z:Z;`
`y,z.`
fragment ***qfrag*****(in** `x:X,` **out** `y:Y)`
pre $P(x)$
post $Q(x,y)$
fragment ***rfrag*****(in** `x:X,` **out** `z:Z)`
pre $P(x)$
post $R(x,z)$.
end template.

Consider for example the fragment ***minmax*** that returns the minimum and maximum values in a list of natural numbers, returning these values in separate output variables.

fragment ***minmax*****(in** `x:L,` **out** `m,n:N)`
pre $x \neq \langle\rangle$
post $m = min(x) \wedge n = max(x)$.

The fragment ***minmax*** can be plugged into the INDEPDECOMP template, replacing the ***main*** fragment. The template is adapted by instantiating the template parameters as follows:

$$
\begin{aligned}
X &\mapsto \mathrm{seq}\,\mathbb{N}\\
Y, Z &\mapsto \mathbb{N}\\
P &\mapsto \lambda\, a \bullet a \neq \langle\rangle\\
Q &\mapsto \lambda\, a, b \bullet b = min(a)\\
R &\mapsto \lambda\, a, b \bullet b = max(a)
\end{aligned}
$$

Applying these parameter instantiations gives rise to the following fragment specifications:

fragment ***qfrag*****(in** `x:L,` **out** `m:N)`
pre $x \neq \langle\rangle$
post $m = min(x)$.
fragment ***rfrag*****(in** `x:L,` **out** `n:N)`
pre $x \neq \langle\rangle$
post $n = max(x)$.

These problems can now be solved separately.

6 Alternative architectures

The final class of adaptation architecture templates are the *alternative architectures*. For this class of the templates the main fragment is implemented by choosing between two or more other fragments. In CARE we consider only deterministic alternatives, where the choice depends on the value of the input arguments of the main fragment.

The CaseAnalysis template lets us implement a fragment (`main`) by breaking the problem into two cases, and implementing these two cases separately. It has the effect of introducing an if-then-else construct. The test is specified by the branching fragment `bfrag`, the if case is represented by the fragment `ifrag` and the else case is represented by the fragment `efrag`. This template is useful when there is a library component that implements the required function for all but some exceptional inputs.

template CaseAnalysis`[`X`;` Y`;` Z`;` $R : X \times Y \to \mathbb{B}$`;` $P : X \times Y \to \mathbb{B}$`;` $Q : X \times Y \times Z \to \mathbb{B}$`]` **is**
fragment `main`**(in** `x:`X**, in** `y:`Y**, out** `z:`Z**)**
  pre $P(x, y)$
  post $Q(x, y, z)$.
`::=` **if** `bfrag(x,y)` **then return** `ifrag(x,y)`
  else return `efrag(x,y)`.
bfragment `bfrag`**(in** `x:`X**, in** `y:`Y**)**
  test $R(x, y)$.
fragment `ifrag`**(in** `x:`X**, in** `y:`Y**, out** `z:`Z**)**
  pre $P(x, y) \land R(x, y)$
  post $Q(x, y, z)$.
fragment `efrag`**(in** `x:`X**, in** `y:`Y**, out** `z:`Z**)**
  pre $P(x, y) \land \neg R(x, y)$
  post $Q(x, y, z)$.
end template

To illustrate the use of the CaseAnalysis template, consider the following example for adding an element to a list of natural numbers, such that the list contains no repetitions both before and after the call to the operation.

type `N ==` $\mathbb{N}$.
type `L ==` $seq\mathbb{N}$.
fragment `addelem`**(in** `e:N`**, in** `s:L`**, out** `t:L`**)**
  pre $NoRep(s)$
  post $NoRep(t) \land elems(t) = elems(s) \cup \{e\}$.

If we attempt to implement this with a call to the fragment `cons` from the List module, then we get the following proof obligations:

$$\forall\, y : \mathrm{seq}\,\mathbb{N} \bullet NoRep(y) \Rightarrow \text{true}$$
$$\forall\, x : \mathbb{N};\ y, z : \mathrm{seq}\,\mathbb{N} \bullet NoRep(y) \land z = x\#y \Rightarrow NoRep(z) \land elems(z) = elems(y) \cup \{x\}$$

For the first proof obligation, we must establish that the precondition of the library fragment `cons` is weaker than that of `addelem` (in effect show that `cons` is defined for any input values that `addelems` is defined for). This proof obligation can easily be proven. For the second proof obligation we must establish that the postcondition of `cons` is stronger than that of `addelem` (we can assume that the precondition of `addelem` holds). However it is easy to find a counterexample to this proof obligation, i.e., the case where $x \in elems(y)$, since in this case the list $z = x\#y$ contains two occurrences of the element x.

Instead we solve this problem by applying the CaseAnalysis template. We replace the fragment `main` by `addelem`, and adapt the template as follows:

$$X \mapsto \mathbb{N}$$
$$Y \mapsto \mathrm{seq}\,\mathbb{N}$$
$$Z \mapsto \mathrm{seq}\,\mathbb{N}$$
$$P \mapsto \lambda\, a, b \bullet NoRep(b)$$
$$Q \mapsto \lambda\, a, b, c \bullet NoRep(c) \land elems(c) = elems(b) \cup \{a\}$$
$$R \mapsto \lambda\, a, b \bullet a \in elems(b)$$

Under this instantiation we get the following specified-only fragments:

bfragment `bfrag`**(in** `x:N`**, in** `y:L`**)**
  test $x \in elems(y)$.
fragment `ifrag`**(in** `x:N`**, in** `y:L`**, out** `z:L`**)**
  pre $NoRep(y) \land x \in elems(y)$
  post $NoRep(z) \land elems(z) = elems(y) \cup \{x\}$.
fragment `efrag`**(in** `x:N`**, in** `y:L`**, out** `z:L`**)**
  pre $NoRep(y) \land \neg\, x \in elems(y)$
  post $NoRep(z) \land elems(z) = elems(y) \cup \{x\}$.

The next stage of the development is to implement the branching fragment `bfrag` by the fragment `isin` from the List module. To do this we instantiate the type parameter from the List modules as follows:

$$E \mapsto \mathbb{N}$$

Next, the fragment `efrag` can be implemented by `cons` from the List module. The following proof obligations must be discharged:

$$NoRep(y) \land \neg\, x \in elems(y) \Rightarrow \text{true}$$
$$NoRep(y) \land \neg\, x \in elems(y) \land z = x\#y \Rightarrow NoRep(z) \land elems(z) = elems(y) \cup \{x\}$$

The first obligation is trivial. The second can be proved by observing that adding an element to a list with no repetitions only produces repetitions if the element was already member of the original list. Finally the fragment `ifrag` can be implemented by returning the second argument, the list y, as an answer (x already appears in y, so we do not want to add it again).

7 Discussion

In CARE, program development starts with a specification of the problem (the *problem specification*). The aim is to implement this problem specification using (one or more) module fragments. However module fragments rarely match the problem specification as-is, so we may employ adaptation templates to adapt the module fragments into a form that satisfies the problem specification. To semi-automate this development process we employ existing formal-based approaches to retrieval. We briefly describe how three formal-based retrieval approaches – signature matching, syntactic matching, and specification (semantic) matching – have been used to find matches between the problem specification and library components. We then briefly discuss how we combine multiple matching steps using tactics.

Signature matching (Rittri 1989, Runciman & Toyn 1989), matches the functional signatures of library components and problem specifications. We employ signature matching prior to syntactic or semantic matching. It serves two purposes. Firstly to narrow the search space by removing any modules or templates with signatures that do not match the

problem specification. Secondly to generate renamings of module/template variables and instantiations of type parameters so that the input/output variables and types of the library module/template fragment are the same as the problem specification fragment.

Syntactic matching (Rollins & Wing 1991, Hemer & Lindsay 2002) is applied when one or both of the fragments includes higher-order parameters. It involves matching specifications using higher-order pattern matching or unification. We employ syntactic matching to match problem specifications against library templates, or to match problem specifications that include parameters (see example below) against modules. Two fragment specifications with respect to some instantiation of the parameters are said to match syntacticly if and only if after applying the instantiation they are the same up to renaming of bound variables. Matches are described in terms of instantiations of parameters. These instantiations may be partial in the sense that not all of the template parameters are instantiated. For example, in Section 6, the parameter R is not instantiated when syntactic matching is applied to match the problem specification (`addelem`) against the main fragment from the CaseAnalysis template. Uninstantiated parameters could either be instantiated in later matches (in this case when matching `bfrag` against `isin`) or instantiated by the user.

When matching against templates that include applicability conditions (such as SeqDecomp from Section 4), we use syntactic matching to calculate parameter instantiations. However we also need to use a theorem prover tool to check that the instantiated applicability conditions are satisfied. In some cases it may be necessary to defer proving the applicability conditions until all of the parameters have been instantiated (as discussed above this can be achieved either by subsequent matching steps or by the user supplying instantiations). An alternative is to prove the applicability conditions in a prover that supports higher-order logic such as Isabelle/HOL (Nipkow, Paulson & Wenzel 2002), but this will require user interaction.

Specification matching (Perry & Popovich 1993, Jeng & Cheng 1995, Zaremski & Wing 1997) is applied when the problem specification and library component specification are first-order (i.e., they do not include any higher-order parameters). Specification matching encompasses a variety of semantic-based matching techniques (Zaremski & Wing (1997) provide a comprehensive listing), where the logical meanings of preconditions and postconditions of fragments (or functions) are compared. We employ specification matching to match problem specifications with no higher-order parameters against modules.

Our current work is focussed on combining multiple matching steps in order to improve the automation of the process. To do this we have defined a tactic language, similar to that used by many interactive theorem provers (Gordon, Milner & Wadsworth 1979), consisting of several basic search tactics and "tacticals" for combining tactics to build more complex tactics. We include a tactical for trying alternate tactics and another for combining tactics in sequence.

With this language we can write tactics that attempt to apply specific templates and modules. Such templates will typically attempt to adapt the problem specification by matching against a sequence of adaptation templates until the problem specification is in a form that can be implemented directly by a module. A very simple tactic that can be easily written in this language is to attempt to match the problem specification against all of the modules (and stop if successful), or first match against the DropInput wrapper template to drop an extra argument then match against all of the modules. This tactic could be used to automatically find implementations for the fragments `ffrag`, `gfrag` and `hfrag` in the first example in Section 4.

8 Related work

The approach described in this paper is similar to the automated component adaptation approach of Penix & Alexander (1997). However our approach differs in several important ways. The first difference is the treatment of higher-order parameters and how they are matched. Penix and Alexander, whose approach is based on the Larch Shared Language (LSL), define parameters in a separate "problem theory" to the main functions. Problem theories give the signature of parameters, and other constructs used in the main theory part. Parameters instantiations are calculated by doing signature matching on problem theories. The separation of parameter definition from the rest of the module results in more false matches than using pattern matching, thus shifting more burden onto semantic based matching.

A second difference is that while Penix and Alexander show a diagram for an architectural component similar to our case analysis template, the corresponding component specification fails to adequately capture this information. In particular while the specification language can capture sequential composition of components, it fails to model branching in component architectures. In contrast our approach can readily capture this information using branching fragments.

Morel & Alexander (2003) use specification slicing to decompose a problem into two or more smaller problem specifications which can be solved individually. Specification slicing is based on data dependencies; in effect they break specifications into independent subproblems. We achieve a similar effect through the use of independent architectures (Section 5), however their approach could be more efficient and could be used as a preprocessing stage.

Bracciali, Bragi & Canal (2002) describe 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 (1999) 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 defined 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.

Konstantas (1993) provides support for interoperability of object-oriented components across different environments and implementation languages. The aims of Konstantas (1993) are similar to the aims of our work, except our focus in more fine-grained. This approach uses type matching (similar to signa-

ture matching) to define relationships between types in order to transform a client interface to a form consistent with the interface of the service.

Finally we say a few words about applying these ideas to other formal component languages. While this paper uses the CARE language to present the ideas of component adaptation schemata, the general ideas are more widely applicable. The closest counterpart to CARE is KIDS (Smith 1990) (now part of Specware), and in general most of the ideas presented in this paper could be applied to enhance the capabilities of KIDS. However one of the main differences between CARE and KIDS is the existence of branching fragments, which as we have already discussed adds much to the expressibility of the language, especially in defining templates/architectures with control branching. Therefore we would lose some of the capabilities if we applied these ideas to KIDS.

The reuse capabilities of other formal languages, in particular state-based languages such as B, could similarly be enhanced, by using adaptation templates. However the existence of state introduces new complexities that need to addressed.

9 Conclusions

In this paper we have described how component-based software engineering can be enhanced by providing support for both component retrieval and component adaptation. The approach described in the paper, using the CARE language and method, builds on existing formal-based approaches to CBSE. In particular we extended existing work on specification matching by defining general strategies for adapting components using templates. Furthermore, we showed how different matching techniques can be combined to provide support for retrieval, where the matching technique used depends on the kind of library component.

Acknowledgements

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

References

Bosch, J. (1999), 'Superimposition : A component adaptation technique', *Information and Software Technology* **4**(5), 249–305.

Bracciali, A., Bragi, A. & Canal, C. (2002), Adapting components with mismatching behaviours, *in* J. Bishop, ed., 'Proceedings of CD'2002', Vol. 2370 of *LNCS*, Springer Verlag, pp. 185–199.

Breymann, U. (2000), *Designing Components with the C++STL*, Addison-Wesley.

Gordon, M. J., Milner, A. J. & Wadsworth, C. P. (1979), *Edinburgh LCF : A Mechanised Logic of Computation*, Vol. 78 of *Lecture Notes in Computer Science*, Springer-Verlag.

Hemer, D. & Lindsay, P. (2002), Supporting component-based reuse in CARE, *in* M. Oudshoorn, ed., 'Proceedings of the Twenty-Fifth Australasian Computer Science Conference', Vol. 4 of *Conferences in Research and Practice in Information Technology*, Australian Computer Society Inc., pp. 95–104.

Hemer, D. & Lindsay, P. (2004), 'Template-based construction of formally verified software', *IEE Proceedings Software* . Accepted for publication.

Jeng, J.-J. & Cheng, B. (1995), Specification matching for software reuse: A foundation, *in* 'Proceedings of ACM Symposium on Software Reuse', pp. 97–105.

Konstantas, D. (1993), Object oriented interoperability, *in* O. M. Nierstrasz, ed., 'Proceedings of ECOOP'93', Vol. 707 of *LNCS*, Springer Verlag, pp. 80–102.

Lindsay, P. & Hemer, D. (1996), An industrial-strength method for the construction of formally verified software, *in* 'Proceedings of ASWEC'96', IEEE Computer Society Press, pp. 26–37.

Lindsay, P. & Hemer, D. (1997), Using CARE to construct verified software, *in* M. Hinchey & S. Liu, eds, 'Proceedings of ICFEM'97', IEEE Computer Society Press, pp. 122–131.

McIlroy, M. (1969), 'Mass produced software components', *Software Engineering Concepts and Techniques* pp. 88–98.

Morel, B. & Alexander, P. (2003), A slicing approach for parallel component adaptation, *in* 'Proceedings of ECBS'03', IEEE Computer Society, pp. 108–114.

Nipkow, T., Paulson, L. C. & Wenzel, M. (2002), *Isabelle/HOL A Proof Assistant for Higher-Order Logic*, number 2283 *in* 'LNCS', Springer.

Penix, J. & Alexander, P. (1997), Toward automated component adaptation, *in* 'Proceedings of the Ninth International Conference on Software Engineering and Knowledge Engineering', pp. 535–542.

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

Rittri, M. (1989), Using types as search keys in function libraries, *in* 'Proceedings of the Fourth International Conference on Functional Programming and Computer Architecture', ACM Press, pp. 174–183.

Rollins, E. & Wing, J. (1991), Specifications as search keys for software libraries, *in* K. Furukawa, ed., 'Eighth International Conference on Logic Programming', MIT Press, pp. 173–187.

Runciman, C. & Toyn, I. (1989), Retrieving re-usable software components by polymorphic type, *in* 'Proceedings of the Fourth International Conference on Functional Programming and Computer Architecture', ACM Press, pp. 166–173.

Schumann, J. & Fischer, B. (1997), NORA/HAMMR: Making deduction-based software component retrieval practical, *in* 'Proceedins of ASE'97', IEEE Computer Society, pp. 246–254.

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

Spivey, J. (1989), *The Z Notation: a Reference Manual*, Prentice-Hall, New York.

Zaremski, A. M. & Wing, J. (1997), 'Specification matching of software components', *ACM Transactions on Software Engineering* **6**(4), 333–369.