

Application Specific Techniques for Retrieval and Adaptation of Trusted Components

Benny Thomas
Master of Computer Science
University of Adelaide, SA, 5005, Australia
Email: benny@cs.adelaide.edu.au

June 16, 2005

Abstract

Component based software engineering (CBSE) is known to save time and money by encouraging software reuse but it is yet to be popularly adopted because of certain largely unresolved problems, including the technical challenges of locating components in the library and adapting them to the specific needs of the user.

Traditional keyword-based retrieval strategies suffer from problems like ambiguity and lack of precision which stem from the use of informal component interfaces. Formal specification matching techniques have been proposed which use mathematically defined pre- and post-condition predicates to define the interface and its components. Then a theorem prover is applied to match the problem specification against library component specifications. These retrieval techniques however are still inadequate as they fail to address the problems of component adaptation and composition.

More recently parameterized templates have been used to define strategies for adapting library components. These templates are used in combination with specification matching to automate component retrieval and adaptation. In this project we propose to develop new templates, and apply the new templates together with existing templates to several case studies to assess how much of the development can be automated.

Keywords: Component based software engineering, Component Retrieval, Adaptation, CARE templates, libraries.

1 Introduction

This is where the introduction will be!!

2 Overview of CARE

CARE (Computer Assisted Refinement Engineering) is a language, toolset and methodology for developing formally verified software.

2.1 The CARE language

The CARE language is functional in nature, with a program consisting of units such as types and fragments. Types are similar to data structures and fragments are similar to functions used in functional programming languages. Each unit is formally specified using a Z-like specification language. Units can either be implemented by calls to other units, or by calls to the primitives from the target code language. A development is said to be complete when all types and fragments have been completed.

CARE supports the notion of reuse by providing a library of *modules* and *templates* both of which consist of a collection of units.

2.2 Types in CARE

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

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

Where $\text{seq}\mathbb{N}$ represents the set of natural numbers. What is to be remembered is that the specification provides an abstract representation of the type and a concrete representation is provided by the implementation.

2.3 Fragments in CARE

There are two possible kinds of fragments in CARE, simple and branching. Simple fragments return a result based on the value of the inputs given. They are defined by an identifier, a list of input and output variables. This is then followed by an precondition (optional) and a postcondition which are used to generate proof obligations that help to establish the correctness of the fragment. For example, consider the fragment *multiply* for multiplying natural numbers. This can be specified as:

```
fragment multiply (in  $a : \mathbb{N}$ , in  $b : \mathbb{N}$ , out  $r : \mathbb{N}$ )  
  pre true  
  post  $r = a \text{ mul } b$ .
```

Branching fragments are used to control flow (they describe which branch should be taken depending on the values of the inputs), but may also return outputs at any of their branches. The simplest branching fragment is defined by giving an identifier, a list of input variables, a precondition (optional) and two or more result branches. Each result branch should include a guard, defining the set of inputs for which this branch will be used, together with an optional postcondition. An example of a simple fragment, the guard of the first branch being a test and the guard for the second branch being the negation of the test, is *iszero*, which tests whether a number is zero.

```
bfragment iszero (in  $x : \mathbb{N}$ )  
  test  $x = 0$ .
```

The test $x = 0$ is the guard for the first branch while the guard for the second branch is the negation of this test i.e. $\neg x = 0$.

Fragments too can be implemented either by calling primitives from the target language directly or by calling other fragments.

2.4 Modules

2.5 Templates

3 Divide and Conquer Template

The basic Divide and Conquer Algorithm consists of the following broad steps:-

- (i) Divide the given instance into sub-instances
- (ii) Solve each of the sub-instances separately
- (iii) Combine the sub-solutions to solve the initial instance

```

template DIVIDE-AND-CONQUERV3.3 [ $X; Y; Z$ ;
 $P : X \mapsto \mathbb{B}, R : X \times Y \times Z \mapsto \mathbb{B}, T : X \times Y \mapsto \mathbb{B},$ 
 $B1 : Z \times Z \times Y \mapsto \mathbb{B}, B2 : Z \times Z \times Y \mapsto \mathbb{B},$ 
 $H : X \times Y \times Z \times Z \mapsto \mathbb{B}, N : Z \times Z \times Y \times Z \mapsto \mathbb{B}]$ 
is
fragment CONQUER(in  $x:X$ , in  $y:Y$ , out  $z:Z$ )
  pre  $P(x) \wedge T(x, y)$ 
  post  $P(z) \wedge R(x, y, z)$ 
  ::= DIVIDE( $x, y$ ) ::  $u, v : Z$ ;
  if  $midpt - cond(u, v, y)$  then COMBINE( $u, v, y$ )
  else if  $less - cond(u, v, y)$  then  $ffrag(u, v, y) :: r : Y$ ;
     $ffrag'(u, v) :: s : Z$ ;
    CONQUER( $s, r$ )
  else  $gfrag(u, v, y) :: r : Y$ ;
     $gfrag'(u, v) :: s : Z$ ;
    CONQUER( $s, r$ )
bfragment  $midpt - cond$  (in  $u : Z$ , in  $v:Z$ , in  $y:Y$ )
  test  $B1(fu(u), y) \wedge B2(fv(v), y)$ 
bfragment  $less - cond$  (in  $u : Z$ , in  $v:Z$ , in  $y:Y$ )
  test  $\neg B1(fu(u), y)$ 
fragment DIVIDE(in  $x:X$ , in  $y:Y$ , out  $u$ , out  $v:Z$ )
  pre  $P(x) \wedge T(x, y)$ 
  post  $P(u) \wedge P(v) \wedge H(x, y, u, v)$ 
fragment COMBINE(in  $u:Z$ , in  $v:Z$ , in  $y:Y$ , out  $z:Z$ )
  pre  $B1(fu(u), y) \wedge B2(fv(v), y) \wedge P(u) \wedge P(v)$ 
  post  $P(z) \wedge N(u, v, y, z)$ 
fragment  $ffrag$  (in  $u : Z$ , in  $v : Z$ , in  $y : Y$ , out  $r : Y$ )
  post  $r = f(u, v, y)$ 
fragment  $ffrag'$  (in  $u : Z$ , in  $v : Z$ , out  $s : Z$ )
  post  $s = f'(u, v)$ 
fragment  $gfrag$  (in  $u : Z$ , in  $v : Z$ , in  $y : Y$ , out  $r : Y$ )

```

```

    post  $r = g(u, v, y)$ 
fragment  $gfrag'$  (in  $u : Z$ , in  $v : Z$ , out  $s : Z$ )
    post  $s = g'(u, v)$ 
fragment  $fu$  (in  $u : Z$ , out  $l : X$ )
    post  $l = fu(u)$ 
fragment  $fv$  (in  $v : Z$ , out  $m : X$ )
    post  $m = fv(v)$ 
end template.

```

4 Insert

Consider for example the fragment INSERT which inserts a natural number into a sorted list of natural numbers, returning a new sorted list with the number inserted.

```

fragment INSERT (in  $x : seq\mathbb{N}$ , in  $y : \mathbb{N}$ , out  $z : seq\mathbb{N}$ )
  pre  $ordered(x) \wedge x \neq \langle \rangle$ 
  post  $ordered(z) \wedge elems(z) = elems(x) \cup \{y\}$ 

```

The fragment INSERT can be plugged into the DIVIDE-AND-CONQUERV3.3 template, replacing the CONQUER fragment. The template is adapted by instantiating the template parameters as follows:

```

 $X, Z \mapsto seq\mathbb{N}$ 
 $Y \mapsto \mathbb{N}$ 
 $P \mapsto \lambda a \bullet ordered(a)$ 
 $T \mapsto \lambda a, b \bullet a \neq \langle \rangle$ 
 $R \mapsto \lambda a, b, c \bullet elems(c) = elems(a) \cup \{b\}$ 
 $B1 \mapsto \lambda a, b \bullet a \leq b$ 
 $B2 \mapsto \lambda a, b \bullet a > b$ 
 $H \mapsto \lambda a, b, c, d \bullet elems(a) = elems(c) \cup elems(d)$ 
 $N \mapsto \lambda a, b, c, d \bullet elems(d) = elems(a) \cup \{c\} \cup elems(b)$ 
 $f \mapsto \lambda a, b, c \bullet c$ 
 $f' \mapsto \lambda a, b \bullet a$ 
 $g \mapsto \lambda a, b, c \bullet c$ 
 $g' \mapsto \lambda a, b \bullet b$ 
 $fu \mapsto \lambda a \bullet last(a)$ 
 $fv \mapsto \lambda a \bullet head(a)$ 

```

After applying these instantiations to the template we get the following fragments that are added to the user program:

```

fragment CONQUER(in  $x:\text{seq}\mathbb{N}$ , in  $y:\mathbb{N}$ , out  $z:\text{seq}\mathbb{N}$ )
  pre  $\text{ordered}(x) \wedge x \neq \langle \rangle$ 
  post  $\text{ordered}(z) \wedge \text{elems}(z) = \text{elems}(x) \cup \{y\}$ 
  ::= DIVIDE( $x, y$ ) ::  $u, v:\text{seq}\mathbb{N}$ ;
  if  $\text{midpt} - \text{cond}(u, v, y)$  then COMBINE( $u, v, y$ )
  else if  $\text{less} - \text{cond}(u, v, y)$  then  $\text{ffrag}(u, v, y) :: r:\mathbb{N}$ ;
     $\text{ffrag}'(u, v) :: s:\text{seq}\mathbb{N}$ ;
    CONQUER( $s, r$ )
  else  $\text{gfrag}(u, v, y) :: r:\mathbb{N}$ ;
     $\text{gfrag}'(u, v) :: s:\text{seq}\mathbb{N}$ ;
    CONQUER( $s, r$ )
bfragment  $\text{midpt} - \text{cond}$ (in  $u:\text{seq}\mathbb{N}$ , in  $v:\text{seq}\mathbb{N}$ , in  $y:\mathbb{N}$ )
  test  $\text{last}(u) \leq y \wedge \text{head}(v) > y$ 
bfragment  $\text{less} - \text{cond}$ (in  $u:\text{seq}\mathbb{N}$ , in  $v:\text{seq}\mathbb{N}$ , in  $y:\mathbb{N}$ )
  test  $\neg \text{last}(u) \leq y$ 
fragment DIVIDE(in  $x:\text{seq}\mathbb{N}$ , in  $y:\mathbb{N}$ , out  $u:\text{seq}\mathbb{N}$ , out  $v:\text{seq}\mathbb{N}$ )
  pre  $\text{ordered}(x) \wedge x \neq \langle \rangle$ 
  post  $\text{ordered}(u) \wedge \text{ordered}(v) \wedge \text{elems}(x) = \text{elems}(u) \cup \text{elems}(v)$ 
fragment COMBINE(in  $u:\text{seq}\mathbb{N}$ , in  $v:\text{seq}\mathbb{N}$ , in  $y:\mathbb{N}$ , out  $z:\text{seq}\mathbb{N}$ )
  pre  $\text{last}(u) \leq y \wedge \text{head}(v) > y \wedge \text{ordered}(u) \wedge \text{ordered}(v)$ 
  post  $\text{ordered}(z) \wedge \text{elems}(z) = \text{elems}(u) \cup \{y\} \cup \text{elems}(v)$ 
fragment  $\text{ffrag}$ (in  $u:\text{seq}\mathbb{N}$ , in  $v:\text{seq}\mathbb{N}$ , in  $y:\mathbb{N}$ , out  $r:\mathbb{N}$ )
  post  $r = y$ 
fragment  $\text{ffrag}'$ (in  $u:\text{seq}\mathbb{N}$ , in  $v:\text{seq}\mathbb{N}$ , out  $s:\text{seq}\mathbb{N}$ )
  post  $s = u$ 
fragment  $\text{gfrag}$ (in  $u:\text{seq}\mathbb{N}$ , in  $v:\text{seq}\mathbb{N}$ , in  $y:\mathbb{N}$ , out  $r:\mathbb{N}$ )
  post  $r = y$ 
fragment  $\text{gfrag}'$ (in  $u:\text{seq}\mathbb{N}$ , in  $v:\text{seq}\mathbb{N}$ , out  $s:\text{seq}\mathbb{N}$ )
  post  $s = v$ 
fragment  $\text{fu}$ (in  $u:\text{seq}\mathbb{N}$ , out  $l:\text{seq}\mathbb{N}$ )
  post  $l = \text{last}(u)$ 
fragment  $\text{fv}$ (in  $v:\text{seq}\mathbb{N}$ , out  $m:\text{seq}\mathbb{N}$ )
  post  $m = \text{head}(v)$ 
end template.

```

4.1 Automation of Insert

The following section will show how the given specification of *insert* can be automated with primitives and other components from the library.

4.1.1 Automation step 1

The branching fragment *less – cond*

```
bfragment less – cond (in  $u : \text{seq}\mathbb{N}$ , in  $v : \text{seq}\mathbb{N}$ , in  $y : \mathbb{N}$ )  
  test  $\neg \text{last}(u) \leq y$ 
```

initially can be modified by applying the DROPINPUT wrapper template, given in Appendix 6.1, to improve its automation. This will result in the dropping of the input v as it is not being used in the fragment. This reduces the branching fragment to

```
bfragment less – cond (in  $u : \text{seq}\mathbb{N}$ , in  $y : \mathbb{N}$ )  
  test  $\neg \text{last}(u) \leq y$ 
```

This can now be solved(automated) using the FUNDECOMP template mentioned in Appendix 6.2

Where,

$$\begin{aligned} f &\mapsto \lambda a, b \bullet a \leq b \\ g &\mapsto \lambda a, b \bullet \text{last}(a) \\ h &\mapsto \lambda a, b \bullet b \end{aligned}$$

The negation of the **test** can be solved in the following manner:

```
bfragment B( $x : X$ )  
  test  $\neg P(x)$   
 ::= if B1( $x$ )  
   then report fail  
   else report pass  
bfragment B1( $x : X$ )  
  test  $P(x)$ 
```

Where $P(x) \equiv \text{last}(u) \leq y$

4.1.2 Automation step 2

The branching fragment *midpt – cond*

```
bfragment midpt – cond (in  $u : \text{seq } \mathbb{N}$ , in  $v : \text{seq } \mathbb{N}$ , in  $y : \mathbb{N}$ )  
  test  $\text{last}(u) \leq y \wedge \text{head}(v) > y$ 
```

is similar to *less – cond*. It can be considered to consist of two individual predicates each solvable using the steps mentioned in 4.1.1 i.e. dropping an input(DROPINPUT) followed by functional decomposition(FUNDECOMP). Where the two predicates are :-

```
 $\text{last}(u) \leq y$  and  
 $\text{head}(v) > y$ 
```

4.1.3 Automation step 3

The fragment *ffrag*

```
fragment ffrag (in  $u : \text{seq } \mathbb{N}$ , in  $v : \text{seq } \mathbb{N}$ , in  $y : \mathbb{N}$ , out  $r : \mathbb{N}$ )  
  post  $r = y$ 
```

can also be easily automated by recursively calling DROPINPUT, refer Appendix 6.3, until only inputs that are being used in the fragment are left. This results in

```
fragment ffrag (in  $y : \mathbb{N}$ , out  $r : \mathbb{N}$ )  
  post  $r = y$ 
```

The automation of this is trivial and can be solved by simple projections. Similarly the fragments *ffrag'*, *gfrag*, *gfrag'* can also be solved.

5 Square root

It is understood that the DIVIDE-AND-CONQUERV3.3 template explained earlier will not match the specifications of all divide and conquer type problems. There might be cases when the number of input parameters do not match and that could result in setbacks to the retrieval of appropriate components. The following example shows how the user specification, which is not an exact match to the given specification of the Divide and Conquer Algorithm shown, can be adapted by simple adaptation steps (i.e. using existing library components) so that it can be solved by the DIVIDE-AND-CONQUERV3.3 template.

Consider the problem of finding the square root of a given number as specified in the fragment SQUARE-ROOT given below. The fragment calculates the square root of a number by sequential approximations and returns the square root of the initial number rounded off to three decimal places. This is however not the most efficient way to find the square root of a number and is used to explain the different applications of the Divide and Conquer template.

```
fragment SQUARE-ROOT(in  $x : \mathbb{R}$ , out  $z : \mathbb{R}$ )
  pre  $x \geq 0$ 
  post  $z \geq 0 \wedge z^2 - 0.001 \leq x \wedge z^2 + 0.001 > x$ 
```

As is obvious this specification does not match the specification of DIVIDE-AND-CONQUERV3.3 template because the number of input parameters do not match. This can be solved by applying the ADDCONSTANTARG template, refer Appendix 6.4, on the SQUARE-ROOT fragment.

The added constant, a , is defined $a : Y$ where $a = 0.001$, as seen in the last postcondition of the SQUARE-ROOT fragment, with *gfrag* being plugged into the fragment CONQUER of DIVIDE-AND-CONQUERV3.3 and the specification for *afrag* is as shown below

```
fragment afrag(out  $y : Y$ )
  pre true
  post  $y = a$ 
```

This results in the following change to the postcondition of SQUARE-ROOT.

```
fragment SQUARE-ROOT(in  $x : \mathbb{R}$ , out  $z : \mathbb{R}$ )
  pre  $x \geq 0$ 
  post  $z \geq 0 \wedge z^2 - 0.001 \leq x \wedge z^2 + y > x$ 
```

The specification for SQUARE-ROOT after applying the ADDCONSTANTARG template can now be rewritten as shown below

```
fragment square-root(in  $x : \mathbb{R}$ , out  $z : \mathbb{R}$ )
  pre  $x \geq 0$ 
  post  $z \geq 0 \wedge z^2 - x \leq 0.001 \wedge z^2 + y > x$ 
      ::= CONQUER( $x$ , afrag);
```

The template is now adapted by instantiating the template parameters as follows:

$$\begin{aligned}
X, Y, Z &\mapsto \mathbb{R} \\
P &\mapsto \lambda a \bullet a \geq 0 \\
T &\mapsto \lambda a, b \bullet a = a \wedge b = b \\
R &\mapsto \lambda a, b, c \bullet c^2 - a \leq 0.001 \wedge c^2 + b > a \\
B1 &\mapsto \lambda a, b \bullet b^2 - a \leq 0.001 \\
B2 &\mapsto \lambda a, b \bullet a = a \wedge b = b \\
H &\mapsto \lambda a, b, c, d \bullet c = a \wedge d = b \\
N &\mapsto \lambda a, b, c, d \bullet d = b \\
f &\mapsto \lambda a, b, c \bullet a \\
f' &\mapsto \lambda a, b \bullet (b + (a/b))/2 \\
g &\mapsto \lambda a, b, c \bullet a = a \wedge b = b \wedge c = c \\
g' &\mapsto \lambda a, b \bullet a = a \wedge b = b \\
fu &\mapsto \lambda a \bullet a \\
fv &\mapsto \lambda a \bullet a^2
\end{aligned}$$

After applying these instantiations to the template we get the following fragments that are added to the user program:

```

fragment CONQUER(in  $x:\mathbb{R}$ ,  $y:\mathbb{R}$ , out  $z:\mathbb{R}$ )
  pre  $x \geq 0$ 
  post  $z \geq 0 \wedge z^2 - x \leq 0.001 \wedge z^2 + y > x$ 
  ::= DIVIDE( $x, y$ ) ::  $u, v:\mathbb{R}$ ;
  if midpt-cond( $u, v, y$ ) then COMBINE( $u, v, y$ )
  else if less-cond( $u, v, y$ ) then ffrag( $u, v, y$ ) ::  $r:\mathbb{R}$ 
    ffrag'( $u, v$ ) ::  $s:\mathbb{R}$ ;
    CONQUER( $s, r$ )
  else gfrag( $u, v, y$ ) ::  $r:\mathbb{R}$ ;
  gfrag'( $u, v$ ) ::  $s:\mathbb{R}$ ;
  CONQUER( $s, r$ )
bfragment midpt-cond(in  $u, v:\mathbb{R}$ ,  $y:\mathbb{R}$ )
  test  $y^2 - u \leq 0.001$ 
bfragment less-cond(in  $u, v:\mathbb{R}$ ,  $y:\mathbb{R}$ )
  test  $y^2 - u > 0.001$ 
fragment DIVIDE(in  $x:\mathbb{R}$ ,  $y:\mathbb{R}$ , out  $u, v:\mathbb{R}$ )
  pre  $x \geq 0$ 
  post  $u \geq 0 \wedge v \geq 0 \wedge u = x \wedge v = y$ 
fragment COMBINE(in  $u, v:\mathbb{R}$ ,  $y:\mathbb{R}$ , out  $z:\mathbb{R}$ )
  pre  $y^2 - u \leq 0.001 \wedge u \geq 0 \wedge v \geq 0$ 

```

```
    post  $z \geq 0 \wedge z = v$ 
fragment ffrag(in  $u, v : \mathbb{R}, y : \mathbb{R}$ , out  $r : \mathbb{R}$ )
    post  $r = u$ 
fragment ffrag'(in  $u, v : \mathbb{R}$ , out  $s : \mathbb{R}$ )
    post  $s = (v + (u/v))/2$ 
fragment gfrag(in  $u, v : \mathbb{R}, y : \mathbb{R}$ , out  $r : \mathbb{R}$ )
    post  $r = u$ 
fragment gfrag'(in  $u, v : \mathbb{R}$ , out  $s : \mathbb{R}$ )
    post  $s = u$ 
end template.
```

6 Appendix

6.1 DROPINPUT template-needs to be modified for branching fragments!

```
template DROPINPUT[X;Y;Z;P : X → ℤ; 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.
```

6.2 FUNDECOMP template-needs to be modified for branching fragments!

The FUNDECOMP template is a template that allows a problem to be functionally decomposed into subproblems that can be solved separately and the results of which can be combined to return a solution to the initial problem. 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 × V → W;
  g : X × Y → U; h : X × Y → 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.

6.3 DROPINPUT

The DROPINPUT fragment is a kind of wrapper template for fragments. Wrapper templates take a single fragment and do some sort of modification on it to provide a new fragment. The DROPINPUT fragment allows a fragment to be implemented by a secondary fragment with one less input. In this case *main* will be implemented by *frag1*.

```
template DROPINPUT[ $X;Y;Z;P : X \rightarrow \mathbb{B}; f : X \rightarrow Z$ ] is  
fragment main(in  $x : X$ , in  $y : Y$ , out  $z : Z$ )  
  pre  $P(x)$   
  post  $z = f(x)$   
 $::= \text{frag1}(x)$ .  
fragment frag1(in  $x : X$ , out  $z : Z$ )  
  pre  $P(x)$   
  post  $z = f(x)$ .  
end template.
```

6.4 ADDCONSTANTARG

This template enables us to pass a constant appearing in the main fragment as an argument in a call to a secondary fragment.

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