SOFTWARE VERIFICATION RESEARCH CENTRE

SCHOOL OF INFORMATION TECHNOLOGY

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 97-03

Reuse of verified design templates through extended pattern matching

David Hemer Peter A. Lindsay

January 1997

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

to appear: *Proc. Formal Methods Europe (FME'97)*, Springer-Verlag 1997.

Reuse of verified design templates through extended pattern matching

David Hemer Peter A. Lindsay

Abstract

CARE provides a framework for construction and verification of programs, based around the recording of reusable design knowledge in parameterized templates. This paper shows how pattern-matching can be used to aid in the selection and application of design templates from a reusable library. A general framework is presented which is independent of the particular matching algorithm used at the level of mathematical expressions. A prototype has been built which supports a large subset of the Z mathematical language.

Keywords: formal methods, program development, refinement, software verification, pattern matching

1 Introduction

1.1 Outline of CARE

Development of formally verified software is often seen as a difficult, time consuming task, requiring somewhat esoteric mathematical skills. The CARE approach [4, 9] attempts to address this problem by providing a library of reusable, pre-proven design templates, which the software engineer can use to develop formally verified programs.

CARE stands for **Computer Assisted Refinement Engineering**. CARE provides a framework within which specification, programming and verification knowledge can be recorded and reused with minimal need for re-proof. The CARE project has been exploring the use of a library of design templates for which most of the difficult parts of modelling and proof have been done once, off-line, by suitably skilled experts. CARE tools then help the user build applications by selecting and instantiating pre-proven refinements to fit the problem at hand, and generating and discharging correctness-of-fit proof obligations. Other CARE tools synthesize compilable source code programs which can be integrated with other system components and tested using common integration testing techniques.

The CARE method is generic and can be tailored for use with different specification languages, programming languages and theorem provers. In particular, it can be used to construct verified software for programming languages which themselves do not have a full formal semantics, by restricting use of target-language code to formally specified library routines which have been verified off-line using techniques appropriate to the target language.

CARE was developed through a collaboration between Telectronics Pacing Systems and the Software Verification Research Centre. Telectronics develops and manufactures software-driven medical devices such as implantable defibrillators. The company has long been motivated to investigate the use of formal methods for the economical and timely development of provably correct software. Specifically, Telectronics had used formal specifications in the development of some of its products, but wanted a method and tools to help verify code and to enable tracing of requirements from specifications through to code and construction of product variants [2]. A grant from the Australian Government enabled more extensive development of the ideas and the construction of a prototype tool-set to support the method.

The prototype tool-set has been populated with a large library of design templates and primitive components for numbers, sets, lists, arrays and records. We have used the CARE method on a number of medium-sized applications including verification of the design of an event logger such as might be used in an embedded device [8]. The tools themselves have been formally specified [3].

1.2 CARE programs

A CARE program consists of types, fragments and theorems. CARE types correspond to data structures; fragments correspond roughly to functions and procedures in a procedural programming language; and theorems correspond to definitions, lemmas and CARE proof obligations (explained below).

Each CARE program component has its own formal specification, which may include constraints on how the component can be used. Program components are classified as *primitive* or *higher-level*. In essence, primitive components are those whose proof of correctness is outside the scope of CARE, while higher-level components have associated proof obligations. More specifically:

Primitive components are supplied as part of the CARE library, and are not written by the ordinary user. Primitive types and fragments are implemented directly in the target programming language and provide access to target language data structures and basic functionality. (B uses a similar approach [7].) The specification of such a component describes the component in terms of a mathematical model of the semantics of the target language and its compiler: a primitive type's specification describes the set of mathematical values corresponding to the associated data structure; a primitive fragment's specification describes the associated target

code's functionality. Primitive theorems are axioms; their statement is their "specification".

Higher-level components are constructed from other components. Higher-level types and fragments express data refinements and algorithm designs respectively, and are implemented in a special-purpose language with a formally-defined mathematical semantics; using this semantics, CARE tools generate proof obligations which show that the component's implementation is correct with respect to its specification (see §3.2 below). Higher-level theorems (lemmas) are "implemented" by proofs.

CARE differs from most other formal software development methods by supporting incremental working – top-down, bottom-up or in a mixture of styles. During development a CARE program may contain components which have specifications but which do not yet have implementations. A complete CARE program is one in which all components are implemented.

1.3 This paper

This paper explains the CARE approach and illustrates some of the main concepts. §2 below introduces the CARE integrated specification and implementation language. §3 outlines how verified programs are developed using CARE. §4 discusses matching at the level of mathematical expressions, §5 extends it to CARE program components, and §6 extends it to whole design templates. §7 illustrates how matching can be used to develop verified programs from a library of pre-verified design templates. §8 discusses ways of improving the effectiveness of library searches by modifying the matching function to take advantage of the semantics of CARE constructs. The examples all use Z-like naming conventions.

2 The CARE language

This section describes the CARE language in more detail. In the rest of this paper, CARE values and types are written in `typewriter` font and mathematical expressions are written in *italics*.

2.1 Mathematical definitions

The mathematical theory part of a CARE program consists of: signatures and axiomatic definitions of constants, functions and predicates; declarations of "generic" (not-further-defined) sorts and definitions of other sorts; and lemmas, with or without their proofs. For example, Fig. 1 shows the definition of a function *append* which appends a value onto the head of a list, and a lemma for calculating the range of an appended list.

Theory definition of function *append*.

$$append : Elem \times \mathrm{seq}\, Elem \rightarrow \mathrm{seq}\, Elem;$$
$$\forall\, h : Elem;\ t : \mathrm{seq}\, Elem \bullet append(h, t) = \langle h \rangle \frown t,$$
$$\forall\, s : \mathrm{seq}\, Elem \bullet \#s \neq 0 \Rightarrow append(head(s), tail(s)) = s.$$

Lemma `ran_of_append`.

$$\forall\, e : X;\ s : \mathrm{seq}\, X \bullet \mathrm{ran}(append(e, s)) = (\mathrm{ran}\, s) \cup \{e\}$$

Figure 1: Example theory components.

2.2 Types

A CARE type declaration consists of a name, a specification and an implementation. The specification is an expression denoting the sort of mathematical values that objects of the type can take. For example, Fig. 4 contains specifications of CARE types for natural numbers, elements and sequences of elements.

Primitive types are implemented by some target language data structure. A higher-level type (the *refined type*) is implemented in terms of one or more other types (the corresponding *concrete types*) by data refinement; the specification describes the relationship between values of the refined type and their concrete representations (the *refinement relation*), an optional condition restricting the values that the refined type may take (the *constraint*), and an optional condition restricting the values the concrete types may take (the *invariant*). An example refined type is given in Fig. 5.

2.3 Fragment specifications

There are two kinds of fragments: *simple* and *branching*. Simple fragments correspond roughly to functions in a procedural programming language; they take inputs and return outputs.[1] Branching fragments differ from simple fragments by also allowing branching of control during execution. A non-standard feature of the CARE language is that branching fragments can return different numbers and kinds of outputs on different branches.

The number and type of inputs taken by a fragment is fixed. The specification of a simple fragment consists of a name, an optional *precondition*, a list of outputs and their types, and the required input/output relationship (or *postcondition*). For example, Fig. 3 gives specifications of the simple fragments `nil`, `car` and `cdr` for manipulating lists, using LISP-like naming conventions.

The specification of a branching fragment consists of its name, an optional precondition and a sequence of guarded branches. Each branch contains a *test*,

[1] CARE has been extended to handle state-changing operations (not treated here).

a description of the outputs and their types, an optional postcondition, and a *report*, which identifies the branch. (The test in the last branch is `true` by default.) Fig. 4 gives examples of specifications of branching fragments `search` and `decompose`. Note that the number and type of outputs on each branch is fixed but may differ from branch to branch. For example, the `search(s,e)` fragment has two cases: when `e` occurs in `s`, it reports `found` and returns an index `i` at which `e` can be found; otherwise it simply reports `notfound` with no outputs. The *guard* of a branch is its test conjoined with the negations of the tests of the preceding branches. For example the guard of the nonempty branch of `decompose(s)` is $\neg\,(\#s = 0)$.

Note that fragment specifications may be under-determined, in the sense that more than one output may satisfy the postcondition for any given input: e.g. `i` in `search(s,e)`. In practice however, the postcondition is often an equation defining the output variables directly as a function of the input variables.

2.4 Fragment implementations

Primitive fragments are implemented by giving code segments in the target language. Higher-level fragments are implemented in terms of calls to other fragments. The CARE implementation language supports the following simple design constructs: assignment of values to local variables, fragment calls, sequencing, branching of control, and data refinement transformations. An *abort* statement is also provided, for use in branches which will never be executed.

Recursive calls and mutual recursion are allowed, provided the recursion eventually terminates. To establish termination, the CARE user supplies a well-founded variant function (or *variant* for short) whose value decreases on recursive calls and is bounded below. Fig. 4 contains example implementations, for branching fragments `search` and `searchAux`.

3 Construction of verified software with CARE

3.1 CARE programs

A CARE program consists of a collection of theories, types and fragments. Components may be specified but not yet implemented. A program is said to be *complete* if all components in the program have been implemented (and in particular, all proof obligations and lemmas have been proven); otherwise it is said to be *partial*.

3.2 Proof obligations

For each higher-level fragment in the program, a CARE tool generates proof obligations that check that the fragment's implementation satisfies its specification. The proof obligations for fragments fall into four categories:

Partial correctness: The result returned at each (non-aborting) leaf of an implementation tree satisfies the appropriate postcondition.

Termination: For recursively-defined fragments, the variant is strictly decreasing on recursive calls.

Well-formedness: For each fragment call, the fragment's precondition (if any) is satisfied.

Non-execution: Execution cannot reach an 'abort' leaf (at least, not for input values which satisfy the fragment's precondition).

For refined types, there are proof obligations to check that the refinement relation defines a function whose domain is given by the invariant and whose range is given by the constraint.

3.3 Templates

A CARE *design template* (or template, for short) is a reusable, parameterised collection of CARE types, fragments and theories, which together encapsulate a piece of design knowledge.[2] Templates can make use of formal parameters as well as textual parameters, ranging over component names. The prototype CARE library contains templates in each of the following categories:

theories: e.g. the theory for ordered sequences - see Fig. 2;

primitives: e.g. a template containing operations for manipulating linked lists, using LISP-like naming conventions - see Fig. 3;

families of algorithms: e.g. a search algorithm for lists - see Fig. 4;

data refinements: e.g. sets implemented as non-repeating lists - see Fig. 5.

Some or all of the types and fragments may have specifications but not implementations: in such cases, the template user is obliged to supply implementations later in the development. Similarly, some or all of the lemmas in the theory part may be assumptions (called *applicability conditions*); the template user is obliged to show that these follow as logical consequences from the definitions already in the CARE program; for example, the template `Ordered Sequences` given in Fig. 2 has two applicability conditions, stating that the ordering relation must be transitive and it must obey the trichotomy law.

Note that in practice, the user may require only part of a template. The CARE template instantiation tool allows the user to indicate what components of the template are of interest; it then determines the complete set of components on which the nominated components depend (including all applicability conditions) and extracts them from the template, appropriately instantiated.

[2] CARE also provides a mechanism for modularisation of templates; space does not permit details to be presented here.

Template `Ordered Sequences` is

Formal parameters: $X, \_ < \_ : X \times X$.

Applicability conditions:
$\forall\, a, b, c : X \bullet a < b \wedge b < c \Rightarrow a < c$,
$\forall\, a, b : X \bullet a < b \vee a = b \vee b < a$.

Theory definition of predicate *isOrderedSeq*.
isOrderedSeq : seq X;
$\forall\, s : \mathrm{seq}\, X \bullet \mathit{isOrderedSeq}(s) \Leftrightarrow \forall\, i : 1 \,..\, \#s - 1 \bullet s(i) < s(i+1)$.

Lemma `singleton_isOrderedSeq`.
$\forall\, e : X \bullet \mathit{isOrderedSeq}(\langle e \rangle)$

Lemma `append_isOrderedSeq`.
$\forall\, h : X;\ t : \mathrm{seq}\, X \bullet \mathit{isOrderedSeq}(\mathit{append}(h, t)) \Leftrightarrow$
$t = \langle\,\rangle \vee (h < \mathit{head}\ t \wedge \mathit{isOrderedSeq}(t))$

end template.

Figure 2: A template containing theory for ordered sequences

Template `Linked Lists` is

Formal parameters: X.

Type `Elem` has specification: X.

Type `LList` has specification: seq X
implementation: *appropriate code for linked list type declarations*

Fragment `nil()` has specification:
output `s:LList` such that $s = \langle\,\rangle$
implementation: *code for the empty list*

Fragment `car(s:LList)` has specification:
precondition $\#s \neq 0$
output `h:Elem` such that $h = head(s)$
implementation: *code for finding the head of the list*

Fragment `cdr(s:LList)` has specification:
precondition $\#s \neq 0$
output `t:LList` such that $t = tail(s)$
implementation: *code for finding the tail of the list*

Fragment `cons(e:Elem,s:LList)` has specification:
output `r:LList` such that $r = append(e, s)$
implementation: *code for appending an element onto the front of a linked list*

Branching fragment `null(s:LList)` has specification:
result defined by cases:
if $\#s = 0$ then report `yes` else report `no`
implementation: *code for checking for the null list*

Branching fragment `decompose(s:LList)` has specification:
result defined by cases:
if $\#s = 0$ then report `empty`
else report `nonempty` with outputs `h:Elem,t:LList`
such that $s = append(h, t)$
implementation:
case `null(s)` of
`yes:` report `empty`.
`no:` report `nonempty` and return `car(s),cdr(s)`.
end template.

Figure 3: Part of a template for linked lists primitives.

4 Matching on mathematical constructs

In what follows (a dialect of) Z is used to model an abstract syntax for the CARE language and matching functions over the language.[3] To start with, a mathematical expression can be either a sort, a formula or a term:

$$MathExpr == Sort \cup Fmla \cup Term$$

Note that mathematical expressions may contain formal parameters.

4.1 Instantiation

A *formal parameter instantiation* indicates the sorts, predicates and functions by which parameters are to be instantiated:

$$FPInst == FParam \twoheadrightarrow \text{seq } Var \times MathExpr$$

We shall write, e.g. $f(x, y) \rightsquigarrow body$ for $f \mapsto (\langle x, y \rangle, body)$. In practice, signatures are also supplied for predicate and function parameters as part of the instantiation, but they will not be modelled here.

The function *instantiate* takes a mathematical expression (the *pattern*) and an instantiation and forms a new mathematical expression (the *target*) by replacing occurences of formal parameters in the pattern in accordance with the instantiation map. The form of instantiation used in CARE is logically sound: the result of instantiating all expressions in a proof is again a proof. The function's signature only is given here; details are straightforward.

$$| \quad instantiate : MathExpr \times FPInst \twoheadrightarrow MathExpr$$

When formal parameters remain in the *target*, the instantiation is referred to as a *partial* instantiation.

Example: The result of instantiating the expression

$$\forall\, a, b : S \bullet P(f(a, b)) \Rightarrow P(f(b, a))$$

with instantiation $S \rightsquigarrow \mathbb{F}\,\mathbb{N}, P(x) \rightsquigarrow \#x = 0, f(x, y) = x \cap y$ is

$$\forall\, a, b : \mathbb{F}\,\mathbb{N} \bullet \#(a \cap b) = 0 \Rightarrow \#(b \cap a) = 0.$$

[3] As presented here, some of the definitions are not type-correct with respect to standard Z, but the explanation should be clear enough to satisfy most readers. $\twoheadrightarrow$ represents finite functions, $\twoheadrightarrow$ partial functions, $\mathbb{F}$ finite power sets, and $\mathbb{P}$ arbitrary power sets.

4.2 Matching

The function *match* can be specified in terms of the *instantiate* function:

$$match : MathExpr \times MathExpr \rightarrow \mathbb{P}\, FPInst$$

$$\forall\, i : match(m_1, m_2) \bullet instantiate(m_1, i) =_\alpha m_2$$

where $=_\alpha$ is α-equivalence: i.e., equality up to renaming of bound variables.

Note that this is an underspecification of matching, in-as-much-as it requires that only matches be returned, but does not strictly require that all possible matches are returned. (A fuller specification might require, for example, that at least one match from each possible equivalence class of matches be returned [5]). The above specification is sufficient for the purposes of this paper however.

Example: Suppose P is a formal parameter ranging over 1-ary predicates, and a ranges over 0-ary functions (constants); then $P(a)$ matches the formula $0 = 0$ in each of the following ways:

$P(x) \leadsto$	$x = x$	$x = 0$	$0 = x$	$0 = 0$
$a \leadsto$	0	0	0	?

Space does not permit a full description of the matching algorithm used by CARE. Basically, it works on structural induction on the pattern, and returns a finite set of instantiations at each step. For example when the pattern is of the form $f(p_1, \ldots, p_m)$, and f is a constant (i.e. a non-parametric function), then the pattern matches only targets of the form $f(a_1, \ldots, a_m)$ such that each p_i matches a_i. Turning this around, the set of matches against $f(p_1, \ldots, p_m)$ can be found by merging (if possible) the sets formed by matching each p_i against a_i, using the following function:

$$mergeInstSets : \mathbb{F}(\mathbb{F}\, FPInst) \rightarrow \mathbb{F}\, FPInst$$

$$mergeInstSets\ \varnothing = \varnothing,\ mergeInstSets\ \{is\} = is$$

$$rest \neq \varnothing \Rightarrow \mathbf{let}\ is_2 = mergeInstSets(rest)\ \mathbf{in}$$
$$\quad mergeInstSets(\{is_1\} \cup rest) = \{i_1 : is_1;\ i_2 : is_2 \bullet mergeInsts(i_1, i_2)\}$$

Two instantiations are mergeable if they agree on their common part:

$$mergeInsts : FPInst \times FPInst \rightarrow FPInst$$

$$mergeInsts(i_1, i_2) = \text{if } (\text{dom}\, i_2) \lhd i_1 =_\alpha (\text{dom}\, i_1) \lhd i_2$$
$$\text{then } i_1 \oplus i_2 \text{ else } \varnothing$$

where $=_\alpha$ stands for element-wise α-equivalence.

These ideas can be extended to give a complete matching algorithm for mathematical expressions. (Complete in the sense that all possible matches are returned, up to α-equivalence and subsetting: see [6] for details for a very similar syntax).

5 Extending matching to CARE program components

5.1 Modelling CARE program components

To extend matching to CARE program components, we need first to model their abstract syntax. For the purposes of this paper, CARE type specifications can be modelled as follows:

TypeSpec

$$name : TypeName$$
$$spec : Sort$$

Fragment specifications can be modelled as follows:

FragSpec

$$name : FragName$$
$$inputvars : VarDeclars$$
$$precond : Fmla$$
$$gsparts : \mathrm{seq}_1\ GSPart$$

Each guarded specification part is modelled as a 4-tuple, consisting of a guard, report, output variables and a post-condition:

$$GSPart == Fmla \times Report \times VarDeclars \times Fmla$$

The input and output variables (together with any local variables in fragment bodies) are modelled as an ordered sequence of variable/type pairs.

$$VarDeclars == \mathrm{seq}(Var \times TypeName)$$

A theorem specification can be modelled as follows:

TheoremSpec

$$name : TheoremName$$
$$statement : Fmla$$

Space does not permit treatment of component implementation here, but the details are straightforward.

5.2 Textual parameters, renaming and instantiation

At the CARE component level, as well as being able to instantiate formal parameters, we also need to be able to rename textual parameters. Using *Renaming*

to stand for such renamings, we can define appropriate functions for performing renamings: e.g. $rename : TypeName \times Renaming \pfun TypeName$. (To preserve meaning, the renaming function will sometimes need to rename bound variables within constructs, to avoid capture of free variables). In what follows, we overload *rename* and extend it to other CARE constructs.

A (CARE-level) *instantiation* thus consists of an instantiation of formal parameters together with a renaming of textual parameters:

$$Inst == FPInst \times Renaming$$

We can now extend the definition of *instantiate* appropriately:

$$\begin{array}{|l} instantiate : TypeSpec \times Inst \pfun TypeSpec \\ \hline instantiate(T, (i, r)).name = rename(T.name, r) \\ instantiate(T, (i, r)).spec = instantiate(T.spec, i) \end{array}$$

5.3 Component-wise matching

This section considers *exact* matching of CARE components; i.e. the situation where, after renaming textual parameters and instantiating formal parameters in the pattern, the result is α-equivalent to the target. (Here α-equivalence means equality up to renaming of variables bound anywhere in the component, including input and output variables.) It turns out that this form of matching is too strict to be much use in practice; in Section 8 we explore useful ways of relaxing the requirements to take advantage of the semantics of CARE constructs components and how they are used.

To start with however, we simply extend the specification of pattern-matching to CARE components in the obvious way: e.g.

$$\begin{array}{|l} match : TypeSpec \times TypeSpec \fun \mathbb{P}\, Inst \\ \hline \forall\, i : match(T_1, T_2) \bullet instantiate(T_1, i) =_\alpha T_2 \end{array}$$

It is now a straightforward matter to extend the matching algorithm for mathematical expressions to CARE constructs. For example to match type specification T_1 against T_2, where $T_1.name$ is a textual parameter, suppose i is a match for sorts ($T_1.spec$, $T_2.spec$) and r is the renaming $\{T_1.name \rightsquigarrow T_2.name\}$; then (i, r) is a match for (T_1, T_2).

Similarly, the algorithm for matching a fragment specification pattern A with a target B proceeds as follows:

1. try to match the input variables and types of A and B;
2. try to match the output variables and types for each branch of A with the output variables in the corresponding branch of B (this only succeeds when there are equal numbers of branches);

3. try to match the guard and postconditions for each branch of A with the guard and postconditions in the corresponding branch of B;

4. finally, try to match the preconditions of A and B.

At each stage of the algorithm, zero or more instantiations are found. The results of these are merged with the instantiations found in the previous stage using a merge function similar to the one given in §4.2, but extended to include renaming of textual parameters. If any stage fails, then the algorithm terminates and returns the empty set of instantiations.

6 Matching and templates

6.1 Instantiation

Templates and programs can be modelled simply as sets of CARE components. To extend the definition of instantiation to templates, we must also include the set of (names of) components in which the user is interested:

$$TempInst == FPInst \times Renaming \times \mathbb{F}\, CompName$$

where

$$CompName == FragName \cup TypeName \cup TheoremName$$

Space does not permit a full definition of the template instantiation function here, but its signature is given by:

$$instantiate : Template \times TempInst \rightarrow ComponentSet$$

The results returned by the template instantiation tool are then processed by the *worksheet manager* which checks that no conflicts will arise. For example, if A is a fully implemented component whose specification agrees with that of a specified-only component B on the worksheet, then A can be added to the worksheet to provide an implementation of B.

6.2 Matching

To search the library for appropriate templates, we supply a *search query*, consisting of a set of component specifications. The search tool then looks for templates which contain components which match any or all of the components in the query. The tool uses the following function:

$$match : Template \times Query \rightarrow \mathbb{P}\, TempInst$$

$$\forall \tau : match(t, q) \bullet instantiate(t, \tau) =_s q$$

where the relation $=_s$ between two component sets holds if for each component specification in one set there is an α-equivalent component specification in the other set.

An algorithm for finding the matches between a template with components $\{t_1, .., t_m\}$ and search query with components $\{q_1, .., q_n\}$ is as follows:

1. Form the set F of all partial surjective mappings from $\{1, .., m\}$ to $\{1, .., n\}$ (since m and n are finite, then F is also finite).
2. For each $f \in F$:
   (a) form the instantiation sets $match(t_j, q_{f(j)})$ for each $j \in \mathrm{dom} f$;
   (b) merge the sets to form a single set of instantiations i_f;
   (c) from i_f, form the set of template instantiations τ_f, by replacing each 2-tuple of the form (i, r), with the 3-tuple
   $$(i, r, \{t_j.name \mid j \in \mathrm{dom} f\})$$
3. Return the union of all τ_f's: i.e. $\cup\{f \in F \bullet \tau_f\}$.

7 Example uses of matching

We illustrate how matching can be applied to a library of design templates to develop CARE programs.

7.1 Development of an algorithm

To illustrate the use of templates and matching, suppose we are given the following specifications for a search problem:

> Branching fragment `find(s:WordList,e:Word)` has specification:
> result defined by cases:
> if $e \in \mathrm{ran}\, s$ then report `found`
> with output `i:Index` such that $s(i) = e$
> else report `notfound`.
>
> Type `Index` has specification: $\mathbb{N}$.
>
> Type `Word` has specification: $Word$.
>
> Type `WordList` has specification: seq $Word$.

An implementation could be developed using the following steps:

Step 1: We begin by giving a library search query containing the above specifications. A match can be found with the template `Linear Search` given in Fig. 4, with renaming $\{$`List` $\rightsquigarrow$ `WordList`, `Element` $\rightsquigarrow$ `Word`, `Index` $\rightsquigarrow$ `Index`$\}$ and formal parameter instantiation $\{E \rightsquigarrow Word\}$.

Step 2: Next we might look for an implementation of `WordList` by supplying a search query containing the specifications of the types `WordList`, `Word` and the fragment `decompose`. A match with the template `Linked List` in Fig. 3 can be found, with renaming $\{$`LList` $\rightsquigarrow$ `WordList`, `Element` $\rightsquigarrow$ `Word`, `decomposeList` $\rightsquigarrow$ `decompose`$\}$ and instantiation $\{Elem \rightsquigarrow Word\}$.

Step 3: We could then use the specifications of `zero`, `increment` and the type `Index` as a search query to find a template containing primitives for natural numbers.

Step 4: Finally, implementations of `Word` and `equal` (for checking equality of list elements) need to be chosen. The choice for `Word` obviously depends on the intended application. As a general rule, templates which introduce a new type would usually define a branching fragment for determining equality; thus, we could expect to find an appropriate implementation included in the same template as chosen for `Word`.

Note that the user has not had to discharge any proof obligations in the above development.

7.2 A data refinement

Fig. 5 gives a template for representing sets as non-repeating lists. To illustrate use of the template, suppose we were given the following specification of an operation for adding a new element to a set:

> Fragment `insert(e:Elem,u:Set)` has
> specification:
> precondition $e \notin u$
> output `v:Set` such that $v = u \cup \{e\}$.

Upon applying the data refinement, matching `insert` with `abstractOperation` and renaming `concreteOperation` to `insertList`, we are left with the problem of implementing the following fragment:

> Fragment `insertList(e:Elem,s:List)` has specification:
> precondition $isNonRep(s) \wedge e \notin \mathrm{ran}\, s$
> output `r:List` such that $isNonRep(r) \wedge \mathrm{ran}\, r = (\mathrm{ran}\, s) \cup \{e\}$.

Template `Linear Search` is
formal parameters: E.
Type `Index` has specification: $\mathbb{N}$.
Type `Element` has specification: E.
Type `List` has specification: seq E.

Branching fragment `search(s:List,e:Element)` has specification:
result defined by cases:
if $e \in \operatorname{ran} s$ then report `found` with output `i:Index` such that $s(i) = e$
else report `notfound`
implementation: `searchAux(s,e,zero)`.

Branching fragment `searchAux(s:List,e:Element,i:Index)` has specification:
result defined by cases:
if $e \in \operatorname{ran} s$ then report `found` with output `j:Index` such that $s(j - i) = e$
else report `notfound`
implementation:
case `decompose(s)` of
`empty:` report `notfound`.
`nonempty:` assign outputs to `h:Element,t:List`;
case `equal(e,h)` of
`yes:`report `found` and return `increment(i)`.
`no:` `searchAux(t,e,increment(i))`.
variant: $\#s$.

Branching fragment `decompose(s:List)` has specification:
result defined by cases:
if $\#s = 0$ then report `empty`
else report `nonempty` with outputs `h:Element,t:List` such that $s = append(h, t)$.

Branching fragment `equal(a,b:Element)` has specification:
result defined by cases: if $a = b$ then report `yes` else report `no`.

Fragment `zero()` has specification:
output `n:Index` such that $n = 0$.

Fragment `increment(m:Index)` has specification:
output `n:Index` such that $n = m + 1$.
end template.

Figure 4: Template for implementation of a list searching algorithm.

Template `Sets As Non-repeating Lists` is

include `Non-repeating Sequences` with $X \rightsquigarrow X$.

Formal parameters: X, $P : X \times \mathbb{F}\, X$, $Q : X \times \mathbb{F}\, X \times \mathbb{F}\, X$.

Type `Element` has specification: X.

Type `List` has specification: $\mathrm{seq}\, X$.

Type `Set` has specification: $\mathbb{F}\, X$
implementation:
value `u:Set` is refined by `s:List` with invariant $isNonRep(s)$
with refinement relation $u = \mathrm{ran}\, s$.

Fragment `abstractOperation(e:Element,u:Set)` has
specification:
precondition $P(e, u)$
output `v:Set` such that $Q(e, u, v)$
implementation:
decompose `u` into `s:List`;
compose `concreteOperation(e,s)` to `v:Set`;
return `v`.

Fragment `concreteOperation(e:Element,s:List)` has
specification:
precondition $isNonRep(s) \wedge P(e, \mathrm{ran}\, s)$
output `r:List` such that $isNonRep(r) \wedge Q(e, \mathrm{ran}\, s, \mathrm{ran}\, r)$.

end template.

Figure 5: Template for data refinement of sets into non-repeating lists, with corresponding refinement of a simple operation on sets.

Template `Non-repeating Sequences` is

Formal parameters: X.

Theory definition of predicate $isNonRep$.
$isNonRep : \text{seq}\, X$;
$\forall s : \text{seq}\, X \bullet isNonRep(s) \Leftrightarrow (\forall i, j : 1 \,..\, \#s \bullet i \neq j \Rightarrow s(i) \neq s(j))$.

Lemma `singleton_isNonRep`.
$\forall e : X \bullet isNonRep(\langle e \rangle)$.

Lemma `append_isNonRep`.
$\forall e : X;\ s : \text{seq}\, X \bullet isNonRep(append(e, s)) \Leftrightarrow isNonRep(s) \wedge e \notin \text{ran}\, s$.

end template.

Figure 6: A template containing theory for non-repeating sequences

But this fragment can be implemented by simply appending `e` onto list `s`:

Fragment `insertList(e:Elem,s:List)` has
implementation: `cons(e,s)`.

From the specification of `cons`, the output `r` of `insertList(e,s)` satisfies $r = append(e, s)$, and correctness of the implementation follows easily from the lemmas `range_of_append` and `append_isNonRep` in Fig. 1 and Fig. 6 respectively.

8 Improvements

This section considers relaxations of the definition of component matching to make library searches more effective.

8.1 Reordering fragment arguments

Note that the order in which input and output variables appear in a fragment's specification is not of great importance: e.g. whether one defines `cons(e:Elem, s:List)` or `cons(s:List,e:Elem)` is largely a matter of taste. Thus one particularly effective improvement is to make the matching function insensitive to the order of inputs and outputs in query fragments and to extend the definition of instantiation to allow reordering of variables. The instantiation function

can then make the corresponding changes to arguments throughout the fragments being instantiated. To achieve this, we modified the syntax of variable declarations in template fragments to remove ordering of arguments:

$$VarDeclars_1 == Var \twoheadrightarrow TypeName$$

The information about desired variable ordering can be added to instantiations by including data of the following type:

$$VarOrdering == \text{seq } Var$$
$$Inst_1 == FPInst \times Renaming \times VarOrdering$$

The definition of the instantiation function for variable declarations becomes:

$$\begin{array}{|l} instantiate_1 : VarDeclars_1 \times Inst_1 \rightarrow VarDeclars \\ \hline instantiate_1(vs, (i, r, p)) = \{j : 1 \mathinner{..} \#p \bullet j \mapsto (p(j), rename(vs(p(j)), r))\} \end{array}$$

The specification of the match function for variable declarations is analogous.

Example: Given input variable declaration set $vs = \{\mathtt{x} : \mathtt{X}, \mathtt{y} : \mathtt{Y}, \mathtt{z} : \mathtt{Z}\}$ renaming $r = \mathtt{X} \rightsquigarrow \mathtt{List}, \mathtt{Y} \rightsquigarrow \mathtt{Element}, \mathtt{Z} \rightsquigarrow \mathtt{List}$ and permutation $p = \langle \mathtt{z}, \mathtt{x}, \mathtt{y} \rangle$, then

$$instantiate_1(vs, (i, r, p)) = \langle \mathtt{c} : \mathtt{List}, \mathtt{a} : \mathtt{List}, \mathtt{b} : \mathtt{Element} \rangle$$

Remark: A slightly more sophisticated generalisation of this approach would be to allow template fragments to have optional arguments; the above solution can be further adapted to cover this case.

8.2 Matching up to other forms of equivalence

Note that, in many cases, the requirement for α-equivalence can be relaxed to "weaker" forms of equivalence: e.g. for formulae, logical equivalence will usually suffice, where it is used in matching preconditions, etc:

$$\begin{array}{|l} match_2 : Fmla \times Fmla \rightarrow \mathbb{P}\, FPInst \\ \hline \forall\, i : match_2(\phi_1, \phi_2) \bullet instantiate(\phi_1, i) \Leftrightarrow \phi_2 \end{array}$$

The CARE semantics can be used to justify the soundness of implementing one component by another component with an equivalent specification.

8.3 Substitution matching on components

When looking for a fragment which implements a given fragment specification, it is often useful to further relax the requirement for equivalence and look for a

fragment which might have a weaker precondition and/or a stronger postcondition than the query fragment. Such a fragment is substitutable for the original in the CARE program without further change, since the resulting proof obligations are weaker than for the original. This leads to the following definition, for example:

$$
\begin{array}{|l}
match_3 : SimpleFragSpec \times SimpleFragSpec \rightarrow \mathbb{P}\, Inst \\
\hline
\forall \tau : match_3(F_1, F_2) \bullet \mathbf{let}\ F = instantiate(F_1, \tau)\ \mathbf{in} \\
\quad F.name = F_2.name \\
\quad F_2.precondition \Rightarrow F.precondition \\
\quad F_2.precondition \wedge F.postcondition \Rightarrow F_2.postcondition
\end{array}
$$

For example, this form of matching would yield `cons` as a possible instantiation of `insertList` in the example in §7.2; the fragment could be implemented by direct instantiation, thereby absolving the user from establishing correctness.

If automated reasoning support is available, the matching algorithm in §6.2 can be adapted to meet the above specification. Note that full logical deduction is not an absolute necessity: even quite weak deductive abilities promise to increase the effectiveness of library template searching. (Generating proof obligations is another possibility). We plan to prototype such an adaptation in the near future by modifying the abstract syntax of formulae.

9 Comparison to other work

The paper by Zaremski and Wing [15] describes how specification matching can be used to compare two components; the application they consider include retrieval for reuse and determination of subtyping relationships. The paper investigates a number of different ways of relaxing the requirement for exact matching, and compares the effectiveness of the resulting search mechanisms. (Note however that our "substitution matching" is not one of those considered.) The framework is extended to matching of modules, which is similar in its goals to our template matching. Their system requires interactive theorem proving support to determine whether or not components match. By contrast our motivation has primarily been to improve browsing of template libraries and "retrieval for reuse", so we have been mainly interested in the case where matching is fully automatable. However, the framework will also support other paradigms which might be content to use of interactive matching (e.g. correctness by construction).

A number of systems are available that perform specification matching at the component level. The Inscape system [10] uses the Inquire predicate-based search mechanism to aid the user in the search for reusable components. The Inquire search mechanism can look for predicates that are equivalent to the query predicate, as well as predicates that are weaker or stronger. The VCR

system [1] uses implicit VDM specifications as queries for retrieval of software components. The search mechanism used searches for components with weaker preconditions and stronger postconditions. Rollins and Wing [12] describe a system, implemented in λProlog similar to our "substitution matching", which is used to match Larch specifications. The search mechanism looks for specifications with weaker preconditions and stronger postconditions.

A restricted form of specification matching is signature matching [11, 13], where properties of the type system can be used to define various forms of matches. This however is not as powerful, or as successful in retrieving desired components, as the above methods using specification matching.

Finally, the reader's attention is drawn to the AMPHION system [14], which makes use of a library of formally-specified FORTRAN routines. AMPHION converts space scientists' graphical specifications into mathematical theorems and uses automated deduction to try to construct and verify a program that satisfies the specification. The success of AMPHION in its particular problem domain is further evidence that reuse of library routines can be made effective with appropriate tool support.

10 Conclusions

This paper has outlined the CARE approach to constructing and formally verifying software, and explored the use of pattern-matching as an aid in the selection and application of design templates from a reusable library. By minimizing the user's need for mathematical inventiveness, both in modelling and proof, CARE is better suited to industrial development of verified software than many methods.

The method is general and can be used in conjunction with a variety of other development methods, both formal and informal. It can be used with a wide variety of specification languages, theorem provers and target languages. It can even be used with programming languages which do not have a full formal semantics. A prototype tool-set has been built [4] which supports a large subset of the Z mathematical language and can synthesize source-code programs in C; the prototype includes a purpose-built automatic theorem prover together with a generic interactive theorem prover extended with CARE-specific tactics and theories.

Acknowledgements:

The authors would like to thank their colleagues on the CARE project, and in particular Keith Harwood who proposed the original approach from which CARE has evolved. Thanks also for the constructive comments made by many of our colleagues at Telectronics and the SVRC.

SVRC technical reports are available by anonymous ftp from ftp.cs.uq.edu.au in the directory /pub/SVRC/techreports.

References

[1] B. Fischer, F. Kievernagel, and W. Struckman. VCR: A VDM-based software component retrieval tool. Technical report, Technical University of Braunschwieg, Germany, November 1994.

[2] K. Harwood. Towards tools for formal correctness. In *The Fifth Australian Software Engineering Conference*, pages 153–158. IREE Australia, May 1990.

[3] D. Hemer and P.A. Lindsay. Formal specification of proof obligation generation in CARE. Technical Report 95-13, Software Verification Research Centre, The University of Queensland, 1995.

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

[5] G.P. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[6] C. B. Jones, K. D. Jones, P. A. Lindsay, and R. Moore. *mural: A Formal Development Support System.* Springer-Verlag, 1991.

[7] K. Lano. *The B Language and Method: A Guide to Practical Formal Development.* FACIT Series. Springer-Verlag, 1996.

[8] P.A. Lindsay. The data logger case study in CARE. In *Proc 5th Aust Refinement Workshop*, 1996. http://www.it.uq.edu.au/conferences/arw96/.

[9] P.A. Lindsay and D. Hemer. An industrial-strength method for the construction of formally verified software. In *Proceedings of the 1996 Australian Software Engineering Conference*, pages 27–36. IEEE Computer Society Press, July 1996.

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

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

[12] E.J. Rollins and J.M. Wing. Specifications as search keys for software libraries. In *Eighth International Conference on Logic Programming*, pages 173–187. 1991.

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

[14] M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, and I. Underwood. Deductive composition of astronomical software from subroutine libraries. In *Proceedings 12th International Conference on Automated Deduction*, pages 341–355, June 1994.

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