

Supporting Component-based Reuse in CARE

David Hemer and Peter Lindsay

Software Verification Research Centre
University of Queensland.
Brisbane. 4072. AUSTRALIA.
Email: {hemer,Peter.Lindsay}@svrc.uq.edu.au

Abstract

The increased reliance on software in critical applications suggests a greater need for formal methods to be used in the development of such software. A number of formal languages and toolsets exist for developing formally specified and verified software; however experience tells us that the development of formally verified software, even with the current tool support, is an expensive process.

By adopting a component-based development methodology, whereby software is developed from reusable components, significant savings can be made. In particular the amount of proof — arguably the most expensive and difficult part of formal development — can be greatly reduced by proving the correctness of reusable components once, off-line, prior to their use.

Tools are required which support the user in adapting and retrieving components from libraries of formally specified components. This paper describes extensions to the CARE toolset that support adaptation and retrieval of reusable components.

1 Introduction

1.1 Composition-based reuse

The idea of constructing programs from *reusable software components* was introduced by McIlroy in 1969 (McIlroy 1969). The idea is analogous to building a complex electronic device from a number of smaller, simpler, well-known components in an electronic engineering context. The engineer browses a catalogue of component descriptions for suitable components which can be pieced together in some manner to build their device. To build their device it may be necessary to modify the catalogue components in some way.

To support component-based reuse, methods and tools must be available that assist the user in adapting and retrieving components. The *adaptability* of a component is a measure of how easy it is to modify a component to solve a particular problem. *Retrievability* is a measure of how easy it is for the user to find a suitable component amongst a library of components. This becomes particularly important once the library of reusable components becomes quite large.

Much of the reuse literature describes informal approaches to the retrieval and adaptation problem (Brown 1996, Krueger 1992), but this is an area where formal approaches have a lot to offer. In theory, formal component specifications should provide an ideal basis for tool support, particularly for parameter matching and instantiation (Zaremski & Wing 1996). The KIDS tool is a good example of reuse of correctness-preserving transformations to solve difficult logistics problems (Smith 1990).

In practice however there has been little reported reuse of more general formally developed components. This paper reports on a toolset developed in an industry/university collaboration to develop high-integrity software from reusable components.

1.2 Supporting software reuse in CARE

The CARE language and toolset was designed to be used by industrial software engineers to build software for safety critical applications (Hemer & Lindsay 1996, Lindsay & Hemer 1997). To help facilitate this a library of reusable *templates* was created, which brought a number of benefits to the overall approach (Lindsay & Hemer 1996). Firstly the fact that they could be proven off-line by a verification expert reduced the amount of formal proof required by the user. Secondly, templates could encapsulate commonly used target code structures, so that the user need not write any target code. Thirdly, the overall development time could be reduced by providing commonly used algorithms and data refinements.

Using CARE in a trial with industrial software engineers, and on case studies, revealed a number of problems. It was difficult to locate applicable templates in the library, and once found, difficult to find the right way of applying the template to the problem. The user often had to adapt their program to suit available templates rather than the other way around.

Enhancing CARE with the ideas presented in this paper has gone a long way towards addressing these problems. Firstly, templates in CARE have been made more flexible by including more adaptation techniques (originally only parameter instantiation and identifier renaming were implemented) and building these into the tool. As a result of this, templates could be applied to a wider variety of problems. The adaptations are all correctness preserving, meaning that pre-proven templates are still semantically correct after adaptation (provided certain applicability conditions are correct).

To assist the user in finding suitable templates, a *retrieval tool* has been built, based on a variety of matching algorithms. The retrieval tool is configurable to user's needs: users may opt between the precision of the search or its efficiency. The retrieval tool also makes use of the semantics of CARE components, and as a result can conduct more "intelligent" searches of the library.

Another advantage of the overall approach is that matching and adaptation have been integrated. This means that when the retrieval tool returns a match between the search query and a template, the appropriate adaptation of the template is also returned. This alleviates the need for the user to perform adaptations, reducing the risk of an incorrect adaptation being performed, which may not be detected until much later in the development.

1.3 Overview of the paper

In Section 2 we give an overview of the CARE language, methodology and toolset. Sections 3 and 4 describe the extensions made to CARE to support reuse. Section 3 describes a tool used for adapting CARE components. Section 4 describes a retrieval tool built on top of a generic search engine. An example, illustrating the use of the CARE toolset, and in particular the adaptation and retrieval tools, is given in Section 5. Section 6 discusses related work, while conclusions appear in Section 7.

2 CARE

CARE (which stands for **Computer Assisted Refinement Engineering**) is a language, toolset and methodology for developing formally verified software. The language and toolset are described briefly in Sections 2.1 and 2.2 respectively. For more details and examples the reader is referred to (Hemer & Lindsay 1996, Lindsay & Hemer 1997).

2.1 Language and concepts

The CARE language is partitioned into three separate levels of constructs - expressions, units and modules. The main features, as found in many formal languages, of each of these separate levels of constructs are listed below.

expressions: include variables, sets, functions, relations, quantifiers and (higher-order) parameters;

units: include operations, data types, definitions and theorems. Units can be parameterised (over both formal and textual parameters). Units have separate specification and implementation parts, meaning that implementation details can be hidden. Units can be implemented either using target code structures, or by calling other units (in particular data and algorithm refinements are supported).

modules: collections of related units can be encapsulated in a single *template* which implements an algorithm, data refinement, theories, or provides access to “primitive” components.

2.1.1 Expressions

The mathematical language used in CARE is a form of higher-order logic, with parameters ranging over functions, relations and sets. The mathematical language is extensible, with the user being able to declare new functions, relations and sets, as well as introducing assertions about the constructs. There is some pre-defined theory, based on the Z Mathematical Toolkit (Spivey 1989).

2.1.2 Units

The next level of components in CARE are *units*, which include types, fragments, operator declarations and assertions. Types correspond to data structures; fragments correspond roughly to functions and procedures in a procedural programming language; and assertions correspond to definitions, lemmas and proof obligations.

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

```
Index ==  $\mathbb{N}$ .  
Element ==  $E$ .  
List == seq  $\mathbb{N}$ .
```

Figure 1: CARE type specifications

```
cons(e:Element, s:List)  
output r:List such that  $r = \text{append}(e, s)$ .  
  
car(s:List)  
pre:  $s \neq \langle \rangle$   
output e:Element such that  $e = \text{head } s$ .
```

Figure 2: Simple fragment specifications

Primitive units are supplied as part of the library, and are not typically 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 (Lano 1996).) The CARE specification of such a component describes the mathematical properties that may be assumed for the component.

Higher-level units are constructed from other units within the CARE framework. Higher-level types and fragments express data refinements and algorithm designs respectively, and are implemented in the CARE language. CARE tools generate proof obligations which show that the unit’s implementation is correct with respect to its specification.

2.1.3 Types

The specification of a CARE type is an expression denoting the set of mathematical values that objects of the type can take. The example given in Fig. 1 contains specifications of CARE types. The first line declares the CARE type **Index**, which is modelled mathematically as the set of natural numbers. The second line declares the type **Element** which corresponds to some given type E . The third line declares the type **List** which corresponds to a sequence of natural numbers.

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.

2.1.4 Fragments

There are two kinds of fragments: *simple* and *branching*. For brevity we omit descriptions of the latter, except to say that branching fragments are a generalisation of simple fragments that allow different types of results to be returned depending on the input.

Simple fragments correspond roughly to functions in a procedural programming language; they take inputs and return outputs. Fig. 2 shows specifications for the simple fragments **cons** and **car** for manipulating lists, using LISP-like naming conventions.

The implementation of primitive fragments and higher-level fragments differ. Primitive fragments are implemented by giving code segments in the target language. The exact nature of the code segments in primitive fragments is dependent on the target language, as well as the code synthesis tools. The current prototype in CARE uses low level C as the target language.

Higher-level fragments are implemented in terms of calls to other fragments. The CARE implementation language supports the following simple design algorithm constructs: assignment of values to local variables, fragment calls, sequencing, branching of control, and data refinement transformations.

The body of a higher-level fragment is tree-structured. Non-branching nodes of the tree correspond to bindings to local variables of the values returned by simple fragment calls or variables. Branching nodes correspond to calls to branching fragments, labelled by the corresponding reports; where branches return values, these values are bound to local variables. Local variables are newly created at the point of assignment, meaning that they are similar to bound variables in quantified expressions. The leaves of the tree define the fragment's output values.

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.

2.1.5 Modules

The module-like structures used in CARE are referred to as *templates*. Templates are parameterised collections of units (fragments, types, assertions etc.) and unit specifications (cf. package headers in Ada), which collectively implement an algorithm, data refinement, or theory, or provide access to primitive components. Templates are typically proven off-line by a proof expert; as part of the proof process, applicability conditions on the parameters are generated which provide sufficient conditions to guarantee a template's correctness.

For example Fig. 3 contains a data refinement of sets in terms of (possibly repeating) lists. For this particular refinement, the set is represented by the range of the list. For example $\langle a, b, a, a, c \rangle$ and $\langle c, a, b \rangle$ both represent the set $\{a, b, c\}$. Note that no invariant is given for this data refinement; in particular repetitions are allowed in the list. The template contains the fragment `abOp1` used for implementing binary operations on sets, and the fragment `abOp2` for performing operations on a set and an element. These set manipulating fragments are all parameterised, so they can be adapted to solve a number of problems. The command “decompose” maps an abstract value to its concrete counterpart, while “compose” maps in the other direction.

2.2 Tool architecture

Fig. 4 shows the architecture of the CARE toolset, including the extensions to the toolsets — the retrieval tool and the template adaptor tool — shown in shaded boxes. The *template adaptor tool*, described in Section 3, extends a previous tool, the template instantiator tool, by including additional adaptation techniques. The *retrieval tool*, described in Section 4, is a new tool. The remainder of the tool and data stores in the diagram are described briefly below. For more details the reader is referred to a paper describing the CARE toolset (Hemer & Lindsay 1996).

Script: The development and verification of a CARE program is driven from a script supplied by the software engineer. A script may include declarations of fragments, types and theories, as well as commands for retrieving and instantiating templates from the library, generating proof obligations, and invoking one of the theorem provers on a given proof obligation.

Parameters:

$E, P1 : \mathbb{F} E \times \mathbb{F} E, Q1 : \mathbb{F} E \times \mathbb{F} E \times \mathbb{F} E,$
 $P2 : \mathbb{F} E \times E, Q2 : \mathbb{F} E \times \mathbb{F} E \times E.$

`Elem == E.`

`Set == $\mathbb{F} E$`

`= value s is refined by l:List`
`with refinement relation $s = \text{ran } l$.`

`List == seq E.`

`abOp1(s:Set,t:Set)`

`pre: $P1(s, t)$`

`output r:Set such that $Q1(r, s, t)$`

`= decompose s into sc:List;`

`decompose t into tc:List;`

`assign conOp1(sc,tc) to rc:List;`

`compose rc into r:Set; return r.`

`conOp1(s:List,t:List)`

`pre: $P1(\text{ran } s, \text{ran } t)$`

`output r:List such that $Q1(\text{ran } r, \text{ran } s, \text{ran } t)$`

`abOp2(s:Set,e:Elem)`

`pre: $P2(s, e)$`

`output r:Set such that $Q2(r, s, e)$`

`= decompose s into sc:List;`

`assign conOp2(sc,e) to rc:List;`

`compose rc into r:Set; return r.`

`conOp2(s:List,e:Elem)`

`pre: $P2(\text{ran } s, e)$`

`output r:List such that $Q2(\text{ran } r, \text{ran } s, e)$`

Figure 3: Sets refined by repeating lists

Worksheet: The current state of the CARE program under development is stored and displayed on a “worksheet”. The worksheet displays the fragments, types and theories that have either been written by the software engineer or gathered from the library, together with all the proof obligations that have been generated. Each component of the worksheet has an associated **status** which indicates the component's standing in the overall development. The worksheet itself is considered complete and correct if and only if all its fragments and types are implemented and all associated proof obligations have been generated and discharged. Note that worksheets are not directly editable by the software engineer: information can only be added to the worksheet or modified via the script.

Library: The library consists of a collection of pre-proven design templates.

Script interpreter: The script interpreter parses the individual script commands and passes annotated fragments, types and theories to the worksheet manager as abstract syntax trees.

Proof obligation generator: Proof obligations are generated purely mechanically from the CARE components and simplified using basic properties of equality, propositional calculus and quantifiers.

Worksheet manager: The worksheet manager controls what goes on the worksheet, where it is placed on the worksheet and with what status. It takes its input from the script interpreter and from the theorem provers, and updates the

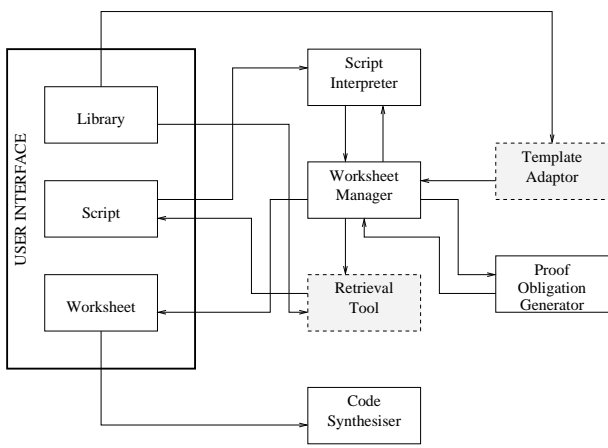


Figure 4: Architecture of the CARE toolset

Component Level	Adaptation Techniques
Expression	Parameter instantiation
Unit	Identifier renaming Variable renaming Argument reordering
Module	Subsetting

Table 1: Component adaptation techniques partitioned into three levels

worksheet accordingly. The worksheet manager is responsible for reporting various errors back to the user via the script interpreter, for example if the user tries to overwrite an already existing implementation.

Code synthesiser: The code synthesiser tool takes a complete collection of fragments and types and constructs a C source-code program.

3 Adaptation tool

The adaptation tool applies an adaptation (from the script) to a library template, returning a set of units. The adaptation tool consists of a number of techniques which are categorised over the three levels of CARE constructs. Table 1 shows the three levels of constructs in CARE and the adaptation techniques applied at each level. These techniques are described separately for each level of constructs below.

The template adaptor tool is an extension of the *template instantiator* tool from the toolset described in (Hemer & Lindsay 1996). The first two techniques described (parameter instantiation and renaming) had already been developed.

A large part of the design and development of templates in CARE is establishing the correctness of the implementation against the specification (e.g. see (Lindsay & Hemer 1996) for a proof of correctness for the accumulator template). It is therefore important that any techniques for adapting components in CARE be *correctness preserving*. Correctness of adaptations is discussed at each structural level below.

3.1 Expressions

For expressions, the main technique is instantiation of formal parameters. Occurrences of parameters within an expression are substituted by other expressions.

Expressions are *instantiated* by replacing occurrences of parameters by other non-parametric expressions. To describe how parameters in an expression are to be replaced, a *formal parameter instantiation* is

given. The instantiation is essentially a finite partial mapping from parameters to expressions such that:

- function parameters are mapped to terms,
- set parameters are mapped to sets,
- relation parameters are mapped to formulae.

The mappings are finite because there are only ever finitely many parameters to instantiate. The mappings are partial indicating that not all parameters need to be instantiated.

In some instances, to show correctness of the parameterised component constraints must be placed on the range of values that the parameters can take. These constraints are referred to as *applicability conditions*. To show that correctness is preserved, the instantiated applicability conditions become proof obligations, that must be discharged by the user.

3.2 Units

For units, the techniques described are renaming of textual parameters, renaming of unit arguments and reordering of unit I/O arguments.

Identifiers can be renamed to achieve meaningful naming within the user’s application domain. To ensure that the correctness of the program is preserved, units must be renamed at the point of definition as well as anywhere that the unit is referenced. Also the identifiers must be renamed to new identifiers which do not already appear within the scope of the renaming.

The number and types of input and outputs for a fragment are described by a variable declaration. The names of these declared variables can be changed without changing the overall meaning of the fragment, provided the changes are done in a consistent manner throughout the unit. Clashes with other local variables are avoided by doing a preprocess renaming of bound variables prior to applying the renaming of I/O arguments.

Another technique is reordering the arguments of fragments. This is more complicated than renaming in that not only must the variables be reordered at the point where they are introduced, but also wherever the fragment is called. This means that the reordering must be applied to any other fragment which calls the fragment in question. Reordering of the I/O arguments of a fragment is correctness preserving provided the arguments of any call to this fragment are similarly reordered.

3.3 Modules

3.3.1 Subsetting

A template consists of a set of formally specified units, some of which may provide optional functionality. Therefore it is often desirable to include subsets of a template. We refer to the adaptation technique where a subset of a template is returned as *subsetting*. The user nominates a subset of units from the template. The adaptation tool calculates the closure of this subset; it is the smallest *self-contained* set of units containing all of the user nominated units. By self-contained we mean that any unit referenced in the set is also included in the set (or at least the specification of the unit).

A template is correct in CARE if each of the units used in the template is at least specified within the template, and each of the non-primitive implemented units is correct with respect to its specification. Therefore the correctness of a template subset follows from the correctness of the entire template, provided that the subset is self-contained.

3.3.2 Parametric polymorphism

Because correctness of primitive units (i.e. those implemented directly by target code constructs) is outside of the scope of CARE, adaptations to target code must be very conservative. Parametric polymorphism is achieved at the code level in a correctness preserving manner by linking the target code data structure with an identifier in the specification. The target code is adapted by giving an identifier renaming.

4 Retrieval tool

4.1 Architecture

The front-end to the CARE retrieval tool sits on top of a generic search engine. The generic search engine is based on a number of algorithms for matching components, decomposed into the three levels of CARE components. The generic search engine is designed in such a way that it can be adapted for a variety of applications. The front-end retrieval tool is the knowledge-based part of the tool, in this case using knowledge of the CARE semantics to build more powerful and flexible searching capabilities.

4.2 Generic search engine

The search engine combines a number of algorithms for matching the different components in CARE. The algorithms are described briefly below for the different levels of components in CARE.

Each matching algorithm takes a component (the *pattern*) and a search *query* (encapsulating the user's requirements), and returns the set of matches. Matches are represented as adaptations of the pattern that satisfy the search query in some manner.

4.2.1 Expression matching

At the expression level two algorithms have been implemented — alpha-equivalence matching and AC-matching. For alpha-equivalence matching, a query q matches a pattern p if there is some instantiation of the parameters in p such that the adapted p is the same as q up to renaming of bound variables. AC-matching (Lincoln & Christian 1990) is a weakening of alpha-equivalence, where the arguments of AC (associative commutative) operators can be reordered. Both of these algorithms have been extended to handle two-way matching (i.e. where both the query and pattern include parameters). For more details the reader is referred to (Hemer 1997).

4.2.2 Unit matching

Matching of units is based on *structural equivalence*; for example (simple) fragments are matched by matching corresponding inputs, outputs, preconditions and postconditions. Only units of the same type will match, e.g. a type will never match a simple fragment, a simple fragment will never match a branching fragment. The query is a unit specification, the pattern may include an implementation (the implementation part is not used in matching however). A query q matches a pattern p if there is an adaptation of the pattern which is structurally equivalent to the query (see (Hemer & Lindsay 1997) for more details). Note that the unit matching algorithms inherit the expression matching algorithms for matching individual expressions such as pre- and postconditions.

4.2.3 Module matching

The search query for module matching consists of a set of unit queries, the pattern is a template. Matching involves matching individual units from the query against units in the template (using the unit matching algorithms). Four different strategies have been implemented: ALL-match in which all query units are matched against template units; SOME-match where at least one query unit must match against a unit(s) from the template; ONE-match where exactly one query unit matches a unit from the template; and HYBRID-match which is a generalisation of the first three strategies. For more details the reader is referred to (Hemer & Lindsay 2001).

4.2.4 Combining the algorithms

These algorithms are combined to form the *search engine*. The search engine is configurable, making it suitable for a number of different applications. The inputs and outputs of the search engine are fairly rudimentary, making it relatively easy to build pre- and post-processing application specific tools.

The main inputs of the search engine are the *search query*, consisting of the specifications of one or more desired units, and a library of pre-proven design templates. A number of other inputs are available including:

interaction level: the user selects one of the interaction levels indicating how much interaction the user has with the search engine. At the lowest level, the search is fully automated, with the searching process completed before any results are outputted. At the highest level of interaction the user is consulted after each match is found and given the option of halting the search process (when a suitable match is found), or continue searching. Other interaction levels consult the user after all matches for a particular template have been compiled; or a guidance mode where matches are outputted as they are found, but where the user does not have the option of halting the process.

type-constrained matching: the user can elect to turn on a type-checker which will eliminate matches that introduce type clashes.

expression-level equivalence: the user selects from alpha-equivalence or AC-equivalence for matching at the expression level.

strategy: the user selects one of the template matching strategies; i.e. all, some, one of hybrid.

The options give the user the choice between precision of the search, and the efficiency of the search. For example turning on type-constrained matching will generally result in a more accurate set of results, but will also slow down the search considerably.

The search engine returns a set of template adaptations corresponding to the set of matches. Note that for a given template there may be multiple matches; these are returned as separate adaptations — some of which may be more useful than others.

4.3 Front-end retrieval tool

The front-end retrieval tool communicates with the search engine. It is responsible for: collecting search information converting it into a form suitable for the search engine; calling the search engine; and outputting the results of the search. These tasks are described below.

4.3.1 Generating inputs

The first stage, driven by an interactive wizard-like GUI, involves collecting search information from the user and the worksheet. This search information includes:

- the names of the worksheet units to be used as basis for building a search query;
- the *matching methods* to be used for each of the nominated worksheet units;
- and a search strategy.

The matching methods available to the user are: *exact matching*, based on matching up to structural equivalence; *relaxed matching*, which uses the semantics of simple fragments to build more intelligence into the search; *branching-alternatives matching* in which the semantics of branching fragments are used to provide a more intelligent search; and *context matching* where implemented worksheet components are used as a means of narrowing the search space. Section 4.4 describes relaxed matching in more detail; for more details on the other matching methods the reader is referred to (Hemer 2000).

This search information is converted automatically by the tool into a form suitable for the search engine. In this stage a search query (consisting of a set of unit specifications), and a search strategy are created from the search information.

The worksheet units nominated by the user form the basis for the search query (i.e., the search is driven by the current state of the program). For exact matching, the worksheet unit is used directly in the query. For context matching, the specification of the worksheet unit is used. For relaxed matching, a new query, more general than the worksheet unit, is created. For branching-alternatives matching, a number of new queries are created, one for each way of ordering the branches in the specification. The search strategy supplied by the user is modified accordingly.

The next stage involves selecting search options and calling the search engine. The search options are passed to the search engine, together with the search query and strategy generated in the previous stage.

4.3.2 Processing outputs

The final stage of the retrieval tool involves displaying the results of the search to the user. Each result consists of a template adaptation - representing a match between the search query - and the template. The result may also contain other units (namely fragment implementations and applicability conditions), associated with unit queries matched used for relaxed or branching alternatives matching. Each result is displayed separately, with the user able to step through the list of results.

4.4 Relaxed matching

For *relaxed matching* the semantics of the CARE language are exploited to match a simple fragment from the worksheet. An alternative to searching for a simple fragment with a specification equivalent to the worksheet unit, is to search for a simple fragment that implements the worksheet fragment.

A simple fragment q (from the worksheet) could be implemented by a fragment p (from a template), by calling the fragment p within the body of the fragment q . Rather than requiring that the specifications of p and q are equivalent (as is the case with *exact*

matching), in this case the pre- and postconditions must satisfy the following relations:

$$q.precond \Rightarrow p.precond \quad (1)$$

$$p.postcond \wedge q.precond \Rightarrow q.postcond \quad (2)$$

These relations are derived from the well-formedness and partial correctness conditions that must be satisfied in order to prove that implementing q with a call to p satisfies the specification of q .

Consider the simple fragment query `addelem` and the specification of a simple fragment pattern `append`:

```
addelem(e:Elem,s>List)
pre: true
output r>List such that ran r = ran s ∪ {e}.

append(e:Elem,s>List)
pre: true
output r>List such that r = ⟨e⟩ ∧ s.
```

The pre-conditions for both the query and pattern are trivial (true). The post-condition for `addelem` is $\text{ran } r = \text{ran } s \cup \{e\}$, the post-condition for `append` is $r = \text{append}(e, s)$. The post-conditions are not logically equivalent, and therefore do not match using exact matching.

However observe that replacing q and p by `addelem` and `append` respectively in (1) and (2) we get

$$\text{true} \Rightarrow \text{true} \quad (3)$$

$$r = \langle e \rangle \wedge s \wedge \text{true} \Rightarrow \text{ran } r = \text{ran } s \cup \{e\} \quad (4)$$

Both of these conditions are clearly satisfiable, therefore `append` is a candidate for implementing `addelem`, using relaxed matching.

Now consider the processing that is performed by the retrieval tool to achieve such a match using relaxed matching. Suppose the user nominates relaxed matching for the specified-only worksheet unit `addelem`. Rather than using `addelem` as part of the search query, the tool creates a new fragment specification that will implement `addelem`. This new unit query is created by replacing the pre- and post-conditions in `addelem` by parameterised formulae:

```
addelem1(e:Elem,s>List)
pre: P(e,s)
output r>List such that Q(r,e,s).
```

Also generated are the following proof obligations that ensure that the matched library fragment provides a correct implementation for `addelem`.

$$\text{true} \Rightarrow P(e, s) \quad (5)$$

$$r = \langle e \rangle \wedge s \wedge P(e, s) \Rightarrow \text{ran } r = \text{ran } s \cup \{e\} \quad (6)$$

Now suppose a search is conducted, and a match is found between the search query containing `addelem1` and a template that includes `append`. Such a match would require the parameters P and Q in `addelem` to be instantiated as:

$$P(e, s) \rightsquigarrow \text{true}, Q(r, e, s) \rightsquigarrow r = \langle e \rangle \wedge s$$

Applying these instantiations to the proof obligations (5) and (6) results in the conditions (3) and (4), which we have already observed are satisfiable.

Matches are displayed to the user in terms of the template adaptations that result in the match. The instantiated proof obligations are also included, as well as an implementation of the original worksheet unit `addelem` in terms of the newly created unit `addelem1`. Supposing the user selects such a match, then the following units are added to the worksheet, together with the instantiated proof obligations (3) and (4).

Dictionary == \mathbb{F} Word.

Word == Word.

```
insert(w:Word,d:Dictionary)
  output r:Dictionary
  such that  $r = \{w\} \cup d$ .
```

Figure 5: Initial design for inserting a word

```
addelem(e:Elem,s>List)
  pre: true
  output r>List such that  $\text{ran } r = \text{ran } s \cup \{e\}$ 
= addelem1(e,s).
```

```
addelem1(e:Elem,s>List)
  pre: true
  output r>List such that  $r = \langle e \rangle \hat{\ } s$ .
```

5 Example

This section illustrates the use of the CARE toolset for developing a simple program. In particular the example illustrates the use of the adaptation and retrieval tools. The example program inserts a word in a dictionary. The dictionary will be represented by a set of words, which in turn will be represented by a list which may contain repetitions of words. This list can in turn be implemented directly in target code using the linked list template from the library.

5.1 Formal specification

Suppose the user wishes to insert a word into a dictionary, they begin by giving the initial specification shown in Fig. 5.

5.2 First refinement step

The first refinement step involves implementing the worksheet units `Dictionary` and `insert`. The following subsections describe the individual steps performed by the user and tools: creating the input to the search engine; calling the search engine; viewing the results and updating the worksheet.

5.2.1 Creating the search input

Firstly the search information is constructed using the current contents of the worksheet. Suppose the user constructs the search information as follows:

- (a) The user chooses the worksheet file containing their program to be the active file.
- (b) The user nominates the *exact* matching method.
- (c) The user selects all of the worksheet units, to be used in constructing the search query.
- (d) The user selects the ALL-match strategy.

The overall search strategy selected by the user is in a sense a default strategy, i.e., doing exact matching on all specified-only units. It may be that the user tries this first, and if it fails then tries other strategies.

The next step involves converting the search information into a form suitable for the search engine. From the fact that exact matching is done on all current worksheet units, the search query consists of the specifications of the units `Word`, `Dictionary` and `insert`. The search strategy to be used is the ALL-match strategy.

```
Dictionary ==  $\mathbb{F}$  Word
= value  $s$  is refined by List
  with refinement relation  $s = \text{ran } l$ .
```

```
List == seq Word
```

```
insert(w:Word,d:Dictionary)
= decompose  $d$  into sc>List;
  assign insertList(sc,w) to rc>List;
  compose rc into r:Dictionary; return  $r$ .
```

```
insertList(s>List,e:Word)
  output r>List such that  $\text{ran } r = \{e\} \cup (\text{ran } s)$ 
```

Figure 7: Worksheet additions first step

5.2.2 Calling the search engine

The user now calls the search engine selecting the interactive search mode. The output from the search engine in interactive mode is shown in Fig 6. Suppose that upon finding a match with the “sets as repeating lists” template (see Fig. 3 on page 3), the user terminates the search.

For this match the query fragment `insert` matches the fragment `ab0p2` from the template, while the query types `Dictionary` and `Word` match the template types `Set` and `Elem` respectively. The formal parameters $P2$ and $Q2$ from the template are instantiated as follows:

$$\begin{aligned} E &\rightsquigarrow \text{Word} \\ Q2(a, b, c) &\rightsquigarrow c = \{b\} \cup a \\ P2(a, b) &\rightsquigarrow \text{true} \end{aligned}$$

To match the template fragment `ab0p2` against the query fragment `insert`, the variables of `ab0p2` are renamed with the mapping $\{s \mapsto d, e \mapsto w\}$ and the input variables are swapped.

5.2.3 Updating the worksheet

After viewing the match results, the user can take the template adaptation and add it to the script. In this case the template adaptation, described above, for adapting the sets as repeating lists template, is added to the script.

The template adaptation is processed by the script editor. As a result, there are a number of additions and modifications to the worksheet, given in Fig. 7. In particular the type `Dictionary` and the fragment `insert` are implemented. Also the specifications for the type `List` and fragment `insertList` are added to the worksheet.

5.3 Second refinement step

The second refinement step involves implementing `insertList`. Firstly the input to the search engine is generated using input from the user and the worksheet. Initially the user might choose to do exact matching on the units `Word`, `List` and `insertList`. However, after failing to find any suitable matches, the user instead elects to do relaxed matching on `insertList`. The user also selects the ALL-match strategy.

Since relaxed matching is used for `insertList`, a new fragment `insertList1`, is created by replacing the pre- and post-conditions of `insertList` with parameterised formulae (as described in Section 4). This newly created fragment is used in the search query.

One of the matches returned by the search tools is with the linked list template, a template that implements primitives for manipulating linked lists.

```

Search_Tool
Match Found with binarychoice...{insert.Dictionary,Word} matched.
Enter command (h for Help):
c.
Match Found with binarychoice...{insert.Dictionary,Word} matched.
Enter command (h for Help):
c.
Searching /home/hemer/CARE_ReleaseV1.0/Library/linearsearch.norm...
Searching /home/hemer/CARE_ReleaseV1.0/Library/sets1.norm...
Match Found with sets1...{insert.Dictionary,Word} matched.
Enter command (h for Help):
c.
Searching /home/hemer/CARE_ReleaseV1.0/Library/sets_as_repeating_lists.norm...
Match Found with sets_as_repeating_lists...{insert.Dictionary,Word} matched.
Enter command (h for Help):
i.
instantiate sets_as_repeating_lists with
  Q2(x1,x2,x3) --> x1 = union(mkSet(x3),x2),
  P2(x1,x2) --> True,
  E --> Word;
  abOp2 --> insert,
  Set --> Dictionary,
  Elem --> Word
include abOp2 with inputs e -> w,s -> d and outputs r -> r.

Enter command (h for Help):
q.
Accept current match (y/n)?
y.
Quitting search tool,0.66
32980
Press return ...

```

Figure 6: Searching the library

```

List == seq Word
= << target code elided.>>

insertList(s>List,e:Word)
= assign insertList1(s,e) to r>List;
  report e and return r

insertList1(s>List,e:Word)
  output r>List such that  $r = \text{append}(e, s)$ 
= << target code elided.>>

Proof obligation
 $\forall r, s : \text{seq Word}; e : \text{Word} \bullet$ 
 $r = \text{append}(e, s) \Rightarrow \text{ran } r = \{e\} \cup \text{ran } s$ 

```

Figure 8: Implementing insert list for repeating lists

Suppose the user adds the adaptation of the linked lists template to the script, which is subsequently processed by the script interpreter. As a result the unit `insertList` is implemented, and a new element `insertList1` is added to the worksheet. Furthermore an applicability condition, associated with relaxed matching of the worksheet unit `insertList` is added to the worksheet as a proof obligation (see Fig. 8). This proof obligation can be easily proven from basic laws associated with sequences.

Fig 9(a) shows the script for the session, while Fig 9(b) shows the resulting worksheet.

5.4 Completing the development

The development is completed by providing an appropriate implementation for `Word`. Such a template is not yet available in the library, but it might be possible to either implement it directly by some target code primitive, or by some refinement in terms of a list of characters. Once a suitable implementation of `Word` is found, the proof obligations can be discharged

and target code generated.

6 Related work

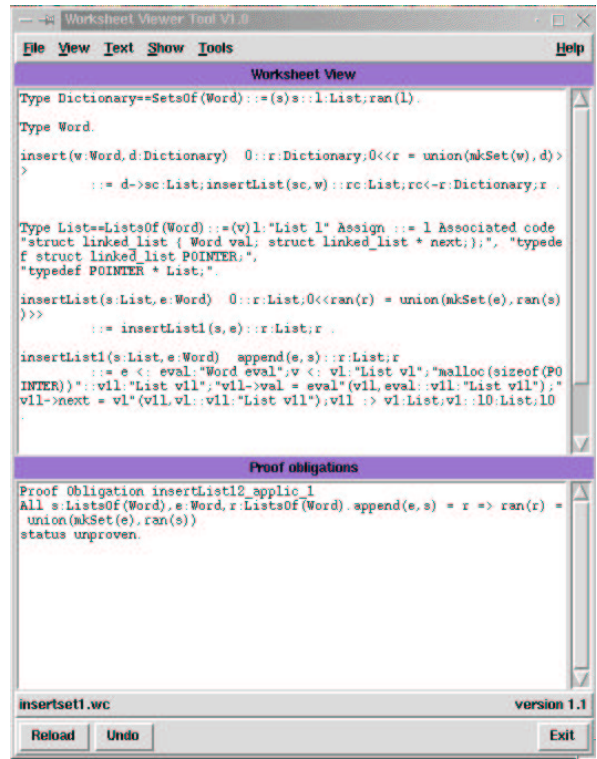
Instantiating formal parameters is a commonly used adaptation technique. Identifier renaming can be thought of as a similar technique to parameter instantiation, however identifiers can only be instantiated to other identifiers. Reordering of the arguments of units is quite different, and to the best of the author's knowledge has not appeared in any of the reuse literature. Similarly, while module subsetting is discussed in the context of component matching by Zaremski and Wing (Zaremski & Wing 1996), it does not appear in the framework of adaptation; consequently the issue of ensuring the subset is self-contained is not raised. Finally the idea of adapting components by changing underlying target-code data structures is similar to Volpano and Kieburtz's (Volpano & Kieburtz 1989) approach, however the approach described in this paper is more general and could be extended to include other kinds of target code adaptation.

The implementation of matching up to AC-equivalence was inspired by an algorithm proposed by Lincoln and Christian (Lincoln & Christian 1990). Type-constrained matching is based on *type-checking*, a technique commonly used to check type consistencies for formal languages.

A number of approaches to matching units with structured functional specifications exist, including *signature matching* (Runciman & Toyn 1989, Rittri 1989, Zaremski & Wing 1995) and *specification matching* (Jeng & Cheng 1995, Zaremski & Wing 1996). Zaremski and Wing (Zaremski & Wing 1996) describe a variety of equivalences for functional specifications, including *exact pre/post match* (similar to structural equivalence), *guarded plug-in* (similar to relaxed matching), *guarded post* etc. Such techniques could easily be incorporated into the front-end tool in a similar manner to relaxed matching.



(a) Script



(b) Worksheet

Figure 9: Development of dictionary insertion

Zaremski and Wing (Zaremski & Wing 1996) describe an approach to module matching, however the approach described here is more general. Firstly, the approach described here allows for a more general unit adaptation framework (beyond just parameter instantiation). Secondly the scope of the kind of units which can appear in modules is extended beyond functions. Thirdly, the Zaremski and Wing approach is restricted to the ALL-match strategy.

The techniques and tools described in this paper could be adapted and applied to other formal languages that support reusable components. KIDS (Smith 1990) supports design tactics that can be adapted by instantiating formal parameters. The Sum language (Bloesch, Kazmierczak, Kearney & Traynor 1995) supports modules which can be parameterised over types and scalar values. Similarly the B language (Lano 1996) support abstract machines which can also be parameterised over types and scalar values. In each case the scope of adaptations could be extended to include techniques similar to those presented in this paper. It would also be possible to build retrieval tools with an architecture similar to the one described here.

7 Conclusions

This paper reports on extensions to the CARE toolset for supporting adaptation and retrieval of reusable components. The techniques and algorithms for adaptation and matching are decomposed into three separate tiers. This has the benefit that additional techniques can be developed at a particular level with minimal changes required to the remaining tool. This decomposition also leads to a highly configurable search engine that can be configured by selecting suitable techniques at each level. The retrieval tool represents one such instance of this configurable search

engine, designed to satisfy the requirements of the CARE methodology.

References

- Bloesch, A., Kazmierczak, E., Kearney, P. & Traynor, O. (1995), 'A methodology and system for formal software development', *International Journal of Software Engineering and Knowledge Engineering* 5(4), 599–617.
- Brown, A. W., ed. (1996), *Component-Based Software Engineering*, IEEE Computer Society, Carnegie Mellon University, Software Engineering Institute.
- Hemer, D. (1997), An algorithm for pattern-matching mathematical expressions, in L. Groves & S. Reeves, eds, 'Proceedings of Formal Methods Pacific'97', Discrete Mathematics and Theoretical Computer Science, Springer Verlag, pp. 103–123.
- Hemer, D. (2000), A Unified Approach to Adapting and Retrieving Formally Specified Components for Reuse, PhD thesis, School of Computer Science and Electrical Engineering.
- Hemer, D. & Lindsay, P. (1996), The CARE toolset for developing verified programs from formal specifications, in O. Frieder & J. Wigglesworth, eds, 'Proceeding of the Fourth International Symposium on Assessment of Software Tools', IEEE Computer Society Press, pp. 24–35.
- Hemer, D. & Lindsay, P. (1997), Reuse of verified design templates, in J. Fitzgerald, C. Jones & P. Lucas, eds, 'Formal Methods Europe '97', number 1313 in 'Lecture Notes in Computer Science', Springer, pp. 495–514.

- Hemer, D. & Lindsay, P. (2001), Specification-based retrieval strategies for module reuse, *in* D. Grant & L. Stirling, eds, 'Proc. of Australian Software Engineering Conference (ASWEC'2001)', IEEE Computer Society, pp. 235–243.
- Jeng, J.-J. & Cheng, B. (1995), Specification matching for software reuse: A foundation, *in* 'Proc. of ACM Symposium on Software Reuse', pp. 97–105.
- Krueger, C. (1992), 'Software reuse', *ACM Computing Surveys* 24(2), 131–183.
- Lano, K. (1996), *The B Language and Method: A Guide to Practical Formal Development*, FACIT Series, Springer-Verlag.
- Lincoln, P. & Christian, J. (1990), Adventures in associative-commutative unification, *in* C. Kirchner, ed., 'Unification', Academic Press, pp. 393–416.
- Lindsay, P. & Hemer, D. (1996), A template-based approach to construction of verified software, Technical Report 96-23, Software Verification Research Centre.
- Lindsay, P. & Hemer, D. (1997), Using CARE to construct verified software, *in* M. Hinchey & S. Liu, eds, 'Proc. 1st Int Conf on Formal Eng Methods (ICFEM'97)', IEEE Computer Society Press, pp. 122–131. SVRC TR 97-40.
- McIlroy, M. (1969), 'Mass produced software components', *Software Engineering Concepts and Techniques* pp. 88–98.
- 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.
- 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.
- 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.
- Volpano, D. & Kieburtz, R. (1989), The templates approach to software reuse, *in* T. Biggerstaff & A.J.Perlis, eds, 'Software Reusability', Vol. 1, Addison-Wesley, chapter 9, pp. 247–255.
- Zaremski, A. & Wing, J. (1995), 'Signature matching: a tool for using software libraries', *ACM Transactions on Software Engineering and Methodology* 4(2), 146–170.
- Zaremski, A. & Wing, J. (1996), Specification matching of software components, *in* 'Third ACM SIGSOFT Symposium on the Foundations of Software Engineering'.