

# A Unified Approach to Adapting and Retrieving Formally Specified Components for Reuse

David Hemer

BSc(Hons), Dip Comp Sc, *Flinders*

March 14, 2000

*Supervisor:* Dr. Peter Lindsay

A thesis submitted for the degree of

Doctor of Philosophy

in the

DEPARTMENT OF COMPUTER SCIENCE AND  
ELECTRICAL ENGINEERING

UNIVERSITY OF QUEENSLAND

## Declaration

I declare that the work presented in this thesis, is to the best of my knowledge original, except as acknowledged otherwise in the text, and that the material has not been submitted, either in whole or in part, for a degree at this or any other university.

David Hemer

Brisbane, September 1999

## Acknowledgements

I would like to acknowledge my supervisor, Dr Peter Lindsay, whose advice, input and support have been invaluable throughout the course of my studies.

*To my darling Megan*

## Abstract

This thesis presents an approach to reusing components which alleviates some of the main problems encountered in component-based reuse; in particular modifying components to suit user's specific needs, and locating suitable components within a library. The focus of the thesis is on components described using a formal language (in other words components with a formal interface specification). The main reason for this is the concise and precise nature of formal languages, which can be exploited in developing more sophisticated methods and tools which take advantage of the semantics of the component.

The solution is presented in two main stages: firstly a framework for adapting components is defined; secondly a framework for retrieving components based on matching component interfaces is defined. Both of these frameworks take advantage of the formal nature of the component interfaces, as a result more sophisticated tools can be developed. For generality it is proposed that formal languages used to represent interfaces are partitioned into three separate levels of granularity - expressions, units and modules - and solutions to adaptation and retrieval are developed separately at each level. An important consideration in developing these frameworks is to ensure that certain component properties are preserved when adapting and retrieving components.

Having proposed these general frameworks, algorithms for adapting and retrieving components are defined in a more concrete and detailed sense within the CARE system. CARE was chosen because the language is relatively simple and compact, yet contains many of the features found in other formal languages, including: variables; functions; predicates; binders; application; typing; parameters; inputs and outputs (and their types); preconditions and postconditions; textual and formal parameters; separation of specification and implementation; case statements; modules; applicability conditions; encapsulation; and information hiding.

These techniques for adapting and retrieving components have been prototyped as extensions to existing CARE tools. As a means of illustrating the value that these extensions have added to the overall CARE system, several example developments using the extended tools are presented at the end of the thesis.

The approach to component reuse presented in this thesis represents a significant advance on other similar approaches. The approach given here is far more general than other approaches, particularly with respect to the scope of components and their interfaces that are considered. Also the adaptation framework goes beyond other approaches which have typically been restricted to parameter instantiation.

# Contents

<b>I</b>	<b>Background</b>	<b>1</b>
<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Software reuse . . . . .	3
1.2	Composition-based reuse . . . . .	5
1.3	Component interfaces . . . . .	7
1.4	Adapting components . . . . .	11
1.5	Retrieving components . . . . .	12
1.6	Integrating adaptation and retrieval . . . . .	15
1.7	This thesis's approach to component reuse . . . . .	16
<b>2</b>	<b>CARE Overview</b>	<b>22</b>
2.1	Introduction . . . . .	22
2.2	Expressions . . . . .	24
2.3	Formally specified units . . . . .	28
2.4	Modular components . . . . .	40
2.5	The CARE methodology . . . . .	47
2.6	The CARE tools . . . . .	49
2.7	Library . . . . .	53
2.8	Discussion . . . . .	57
<b>II</b>	<b>Adaptation</b>	<b>58</b>
<b>3</b>	<b>Adapting Components for Reuse</b>	<b>60</b>
3.1	Introduction . . . . .	60
3.2	A systematic approach . . . . .	61
3.3	Adaptation examples . . . . .	63
<b>4</b>	<b>A Structured Approach to Adaptation</b>	<b>71</b>
4.1	Introduction . . . . .	71
4.2	Adapting expressions . . . . .	72
4.3	Adapting units . . . . .	80
4.4	Adapting modules . . . . .	89
4.5	Example - Summing the elements of a list . . . . .	95
4.6	Discussion . . . . .	99

---

<b>III</b>	<b>Retrieval</b>	<b>103</b>
<b>5</b>	<b>A Structured Approach to Component Retrieval</b>	<b>105</b>
5.1	Introduction . . . . .	105
5.2	A general structured approach to retrieval . . . . .	106
5.3	Retrieval examples . . . . .	112
5.4	Development of a retrieval tool in CARE . . . . .	116
<b>6</b>	<b>Matching Expressions</b>	<b>119</b>
6.1	Introduction . . . . .	119
6.2	Specification of matching algorithm . . . . .	121
6.3	Basic algorithm for matching . . . . .	121
6.4	Proof of correctness for match algorithms . . . . .	131
6.5	Associative commutative matching . . . . .	140
6.6	Type-constrained matching . . . . .	145
6.7	Interactive matching . . . . .	150
6.8	Unification . . . . .	153
6.9	Discussion . . . . .	156
<b>7</b>	<b>Matching Units</b>	<b>158</b>
7.1	Introduction . . . . .	158
7.2	A basic approach to unit matching . . . . .	160
7.3	Extending the matching algorithm . . . . .	171
7.4	Discussion . . . . .	174
<b>8</b>	<b>Matching Modules</b>	<b>176</b>
8.1	Introduction . . . . .	176
8.2	Requirements analysis . . . . .	178
8.3	An algorithm for matching modules . . . . .	183
8.4	Matching templates . . . . .	192
8.5	A generic search engine . . . . .	193
8.6	Discussion . . . . .	196
<b>IV</b>	<b>Applications</b>	<b>197</b>
<b>9</b>	<b>A Library Retrieval Tool</b>	<b>199</b>
9.1	Introduction . . . . .	199
9.2	Automating CARE program development . . . . .	201
9.3	Example - reversing a list . . . . .	211
9.4	Inserting a word in a dictionary (with repetitions) . . . . .	230
9.5	Inserting a word in a dictionary (without repetitions) . . . . .	238
<b>10</b>	<b>Conclusions</b>	<b>249</b>
10.1	Summary . . . . .	249
10.2	Further Work . . . . .	253

<b>A</b>	<b>Predefined Mathematical Constructs in CARE</b>	<b>259</b>
<b>B</b>	<b>CARE tool syntax</b>	<b>261</b>
B.1	Simple fragments . . . . .	261
B.2	Branching fragments . . . . .	262
B.3	Fragment specifications . . . . .	263
B.4	Fragment implementations . . . . .	266
B.5	Types . . . . .	268
<b>C</b>	<b>A Partial Library of CARE Templates</b>	<b>269</b>

# List of Figures

1.1	A 3-tiered classification scheme for reusable software components . . .	17
2.1	Example assertions in CARE . . . . .	31
2.2	Example CARE type specifications . . . . .	31
2.3	Refined type . . . . .	33
2.4	Simple fragment specifications . . . . .	35
2.5	Branching fragment specifications . . . . .	36
2.6	Part of a CARE program for reversing a list. . . . .	38
2.7	A fragment for decomposing lists. . . . .	39
2.8	The Accumulator template . . . . .	42
2.9	Formal parameters for the accumulator template . . . . .	44
2.10	Applicability conditions for the accumulator template . . . . .	45
2.11	Theory for the accumulator template . . . . .	45
2.12	Types for the accumulator template . . . . .	45
2.13	Fragments for the accumulator template . . . . .	46
2.14	Architecture of the CARE toolset . . . . .	49
2.15	The script interpreter for the reverse example . . . . .	51
2.16	The worksheet manager for the reverse example . . . . .	53
2.17	Applicability conditions for the associative commutative accumulator . . . . .	54
2.18	Refining sets as (possibly) repeating lists . . . . .	55
2.19	Refining sets in terms of non-repeating lists . . . . .	55
2.20	Refining sets in terms of ordered lists . . . . .	55
3.1	The map function in Standard ML . . . . .	64
3.2	Renaming the variables in the map function . . . . .	64
3.3	Reordering the arguments of map . . . . .	65
3.4	A generic schema for representing a store inventory . . . . .	65
3.5	A generic class for vectors in C++ . . . . .	67
3.6	Manipulating complex numbers in C++ . . . . .	67
3.7	An Isabelle theory for natural numbers . . . . .	69
3.8	An instance of the natural numbers theory . . . . .	69
3.9	A class for stacks in Object-Z . . . . .	70
4.1	The fragment gimli prior to variable renaming . . . . .	83
4.2	The fragment gimli after variable renaming has been performed . . . . .	83
4.3	Fragment collection prior to reordering of input/output . . . . .	86

4.4	Fragment collection after reordering of input/output variables . . . . .	87
4.5	A list of naturals implemented in terms of a linked list of unsigned integers . . . . .	92
4.6	Implementing natural numbers as unsigned integers . . . . .	92
4.7	A polymorphic linked list . . . . .	93
4.8	An adaptation of a polymorphic linked list . . . . .	93
4.9	Initial specification for summing a list of numbers . . . . .	96
4.10	Adapting the accumulator template . . . . .	96
4.11	Implementing <code>sumList</code> using the accumulator template . . . . .	97
4.12	Adapting the unsigned integers template . . . . .	98
4.13	Implementing natural number primitives using unsigned integers . . . . .	98
4.14	Adapting the linked lists template . . . . .	99
4.15	An adaptation of the linked list template . . . . .	100
5.1	A query for searching Isabelle theories . . . . .	113
5.2	The subset and equality relations in Isabelle . . . . .	114
5.3	A search query in $Z$ . . . . .	115
5.4	A generic schema for representing a store inventory . . . . .	115
5.5	A search query for Object- $Z$ . . . . .	116
5.6	A class for stacks in Object- $Z$ . . . . .	117
7.1	A simple fragment query and matching candidate . . . . .	164
7.2	Matching type specifications . . . . .	168
7.3	Matching operator declarations . . . . .	170
7.4	Matching assertions . . . . .	171
7.5	Matching fragment specifications . . . . .	174
8.1	A search query for finding natural number primitives . . . . .	181
8.2	Equivalent ways of specifying integer division . . . . .	183
8.3	A search query for matching templates . . . . .	193
8.4	A match with the reverse query . . . . .	193
9.1	Using relaxed matching of fragments . . . . .	205
9.2	Equivalent branching fragment specifications . . . . .	206
9.3	A specification of a simple fragment from a worksheet . . . . .	208
9.4	Creating a new search query from a worksheet unit . . . . .	208
9.5	Implementing the worksheet fragment <code>f1</code> by the new fragment <code>f2</code> . . . . .	208
9.6	A branching fragment with three branches . . . . .	209
9.7	A replacement specification for <code>bf</code> . . . . .	209
9.8	An implementation of <code>bf</code> by <code>bf1</code> and associated applicability conditions . . . . .	210
9.9	An initial script for reversing a list of natural numbers . . . . .	212
9.10	Initial design for reversing a list of natural numbers . . . . .	213
9.11	Branching alternatives for <code>decompose</code> . . . . .	215
9.12	Units associated with the search query unit <code>decompose1</code> . . . . .	216
9.13	Units associated with the search query unit <code>decompose2</code> . . . . .	216

---

9.14	Constructing search information . . . . .	217
9.15	Searching the library . . . . .	219
9.16	Viewing the search results . . . . .	221
9.17	Script commands after first refinement step . . . . .	222
9.18	Reverse program after first refinement step . . . . .	223
9.19	Creating search information . . . . .	225
9.20	Searching the library during second refinement step for reverse . . . . .	226
9.21	Search results for second refinement step of reverse program . . . . .	227
9.22	Script commands after the second refinement step . . . . .	228
9.23	Reverse program after second refinement step . . . . .	229
9.24	Low level C code generated for the reverse example . . . . .	231
9.25	Initial design for inserting a word into a dictionary . . . . .	232
9.26	Creating the search information . . . . .	234
9.27	Searching the library . . . . .	235
9.28	Viewing the search results . . . . .	236
9.29	Worksheet additions after refining sets in terms of repeating lists . . . . .	237
9.30	Creating the search information . . . . .	239
9.31	Viewing the search results . . . . .	240
9.32	Implementing insert list for repeating lists . . . . .	241
9.33	A script for the dictionary insertion program . . . . .	242
9.34	A worksheet for inserting a word in a dictionary . . . . .	243
9.35	First refinement step . . . . .	244
9.36	Second refinement step . . . . .	246
9.37	Third refinement step . . . . .	248
C.1	The Accumulator template . . . . .	271
C.2	The Associative Commutative Accumulator template . . . . .	272
C.3	The if then intro template . . . . .	273
C.4	Linked lists . . . . .	274
C.5	Natural numbers . . . . .	275
C.6	Sets refined by (possibly) repeating lists . . . . .	276
C.7	Sets as non-repeating lists . . . . .	277
C.8	Sets as ordered lists . . . . .	278

