Available at www.ElsevierComputerScience.com POWERED BY SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

ELSEVIER Electronic Notes in Theoretical Computer Science 91 (2004) 116–133

www.elsevier.com/locate/entcs

Higher-order associative commutative pattern matching for component retrieval

David Hemer[1]

School of Information Technology and Electrical Engineering
The University of Queensland

Abstract

In this paper we describe a higher-order associative commutative pattern matching algorithm. We are motivated by the need for developing tool support for matching user requirements against library component interfaces, both specified using a formal language. In developing such tool support we aim for a maximum level of recall, while at the same time maintaining a reasonable level of automation and efficiency.
In order to support adaptation of library components we assume the library components may contain higher-order parameters (representing types, functions and relations) — components are adapted by instantiating parameters to suit the requirements of the user. However with this assumption, the usual specification matching techniques, based on proving equivalence using a theorem prover based on first-order logic, are no longer useful in general. We therefore propose building tool support based on pattern matching, and increasing recall by providing (partial) support for matching expressions that can be shown to be equivalent by applying the laws of associativity and commutativity.

Key words: Associative commutative matching, specification matching, component retrieval

1 Introduction

1.1 Motivation

Component-based software engineering (CBSE), originally proposed by McIlroy in 1969 [11], is a development paradigm where software is built by piecing together a number of software building blocks, referred to as *components*. The

[1] Email: hemer@itee.uq.edu.au

doi:10.1016/j.entcs.2003.12.008

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 to build their device.

Research into component-based software engineering has been re-invigorated in recent times, with particular focus on formal approaches to CBSE [17,6,1,20,19]. In particular there has been an emphasis on formalising component interfaces, and using this extra expressiveness and preciseness to develop support for adapting and retrieving components to meet user requirements.

The most commonly used formal-based mechanism for supporting component adaptation is the use of (formal) higher-order parameters within library components. Such components can be adapted by instantiating the higher-order parameters to types, terms or predicates in order to meet user requirements. As we will explain below, the use of higher-order parameters has implications on the kind of retrieval support that can be developed.

In terms of retrieval, a promising approach is to formally specify the behaviour of library components and the user requirements, and use a technique referred to as *specification matching* [22] to retrieve library components that satisfy the user requirements. Specification matching involves checking that a library component satisfies the user requirements; both library components and user requirements are formally specified using a component language.

1.2 Formal approaches to component retrieval

Specification matching is based on *matching* two component specifications, representing the component library specification (the *pattern*), and the user requirements (the *query*). For the purpose of this paper we will consider specification matching techniques in two broad categories: *syntactic matching*; and *semantic matching*.

Syntactic matching techniques [16,5] are based on pattern matching or unification. The advantage of these techniques is that they are generally quite efficient and can be fully automated. Furthermore unification and pattern matching can be applied to higher-order logics, and are therefore compatible with our aim of supporting adaptation of components through the use of higher-order parameters. The main disadvantage of unification based approaches is the poor level of *recall* associated with purely syntactic matching.

Semantics matching techniques [22,9,15,18] are based on *proving* that library components satisfy user requirements. The main advantage of semantic-based matching techniques is their increased level of recall. Indeed semantic-based specification matching can be extended to include *relaxed* matching techniques. For example, semantic-based techniques can be used to match a query against a library component with a weaker pre-condition and stronger post-condition. However semantic matching techniques require reasoning support (usually in the form of an interactive theorem prover), which can become a major overhead in the retrieval process, both in terms of efficiency and the level of automation. Furthermore, the current approaches to specification matching are all based on first-order logics, so such techniques are inconsistent with our goal of supporting adaptation through the use of higher-order parameters.

1.3 This paper

The aim of this paper is to present a solution to the problem of specification matching that sits somewhere between the two categories of matching techniques described above. We describe a higher-order pattern matching based technique, which is knowledge based in the sense that it can apply associativity and commutativity rules in order to increase the level of recall.

In Section 2 we summarise existing approaches to pattern matching and unification of higher-order logics, including existing approaches to associative commutative matching of first order logics. In Section 4 we define our expression language, give a description of parameter instantiation, and describe a purely syntactic pattern matching algorithm for the expression language. In Section 5 we give a specification for associative commutative matching. In Section 6 we define an (incomplete) associative commutative pattern matching algorithm for higher-order expressions. In Section 7 we discuss how this matching technique can be used to support component retrieval.

2 Related work

The Prolog language contains first-order (untyped) meta-variables. Deduction in Prolog is based on so-called standard (first-order) unification of these meta-variables. Instances of terms are obtained by substituting one or more of the variables in the term with new subterms, e.g. $f(a, b)$ is an instance of $f(x, b)$ where the subterm a is substituted for the meta-variable X. Two terms are said to unify if they have a common instance, where variables common to both terms are substituted consistently.

In the case of higher-order logics (i.e. logics whose parameters range over functions or relations), standard unification is no longer applicable. Huet [8], was first to propose an algorithm for unifying higher-order terms in typed lambda-calculus. Since then many implementations of Huet's algorithm have been developed. It has been established that in general higher-order unification is undecidable (that is given arbitrary terms it is not always possible to determine when they are unifiable). However the unification of certain classes of higher-order terms have been found to be decidable. In particular for a class of terms discovered by Miller referred to as "patterns", unification has been shown to be decidable [12,13]. Similarly third-order matching has been shown to be decidable [2], and there are certain classes of higher-order terms for which matching is decidable [21].

Higher-order unification has been most commonly applied to automated reasoning. The HOL theorem prover [4] is "hardwired" to higher order logic, and provides built-in support for higher-order unification. On the other hand, the Isabelle theorem prover [14] is a generic theorem prover, designed for reasoning in a variety of formal theories. It does however provide support for higher-order logic, with a built-in algorithm for higher-order unification. Note that its (lazy) algorithm is not guaranteed to terminate, since the set of "most general unifiers" for terms in its metalanguage is not always finite.

Higher-order unification has also been utilised in programming languages. As an example lambda-Prolog extends standard Prolog, by allowing parameters to range over functions and relations. As a result reasoning in lambda-Prolog relies on higher-order unification rather than standard unification.

Finally, we note that associative commutative matching and unification has been well studied for first-order logics. In particular we take inspiration from an algorithm developed by Lincoln and Christian [10].

3 Example

We begin by looking at a motivating example, which while simple, cannot be handled by standard pattern matching, or semantic-based matching techniques. Suppose the user requires a function for adding an element to a list such that the list contains no repetitions before and after the element has been added, and the list never has more than 50 elements. Such a requirement could be represented with the following specification:

addelem(**in** *e:E*, **in** *s:List*, **out** *r:List*)
  pre $isNonRep(s) \wedge len(s) \leq 50$
  post $ran(r) = ran(s) \cup \{e\} \wedge len(r) \leq 50 \wedge isNonRep(r)$.

This problem can be solved by doing case analysis on the whether or not e is a member of the list s. This can be achieved using a case analysis library component. The specification of the top-level function in this library component is shown below, where P and Q are parameters.

cases(**in** *x:X*, **in** *y:Y*, **out** *z:Z*)
  pre $P(y)$

post $P(z) \wedge Q(x, y, z)$.

Therefore to solve the problem, we need to find a match between `addelem` and `cases`. Using *signature matching*, we can easily align the names and types of the input/output arguments of the query and library specification, i.e., the library specification can be adapted to give:

cases(**in** *e:E*, **in** *s:List*, **out** *r:List*)
  pre $P(s)$
  post $P(r) \wedge Q(e, s, r)$.

However simple pattern matching will not return a match because the conjuncts in the postcondition of the library specification are in a different order to those of the query. The AC matching algorithm described in this paper does return a match for this problem, instantiating $P \mapsto \lambda\, x \bullet \mathit{isNonRep}(x) \wedge \mathit{len}(x) \leq 50$, $Q \mapsto \lambda\, x, y, z \bullet \operatorname{ran}(z) = \operatorname{ran}(y) \cup \{x\}$.

4 Higher-order terms

4.1 Abstract syntax

We begin by defining the mathematical expression language used in this paper. The sets *Var*, *FunctionName* and *FunctionParam* represent the name sets for variables, function names and function parameters respectively. In this paper, variables will be represented by lower case identifiers u, v, w, x, y and z. Function names are represented by lower case identifiers f, g and h for functions of arity 1 and greater, and the identifiers c, d and e for constants. Function parameters are represented by the upper case identifiers F, G, and H.

$$[\mathit{Var}, \mathit{FunctionName}, \mathit{FunctionParam}]$$

A term can be either a *placeholder*, a *variable*, a *(non-parametric) function* application or a *parametric function* application. Placeholders are used to define instantiations of function parameters, they are represented abstractly as a natural number. For example given an instantiation of a parameter F to the term $\operatorname{fnapplic}(g, \langle \mathit{ph}(2), \mathit{ph}(1)\rangle)$, means that an application of F (which must have at least two arguments) is replaced by the function g applied to the second and first arguments of F.

Functions map a number of values (the arguments) to a single value; an application of a function is represented by a function name and a sequence of terms. Terms can be parameterised over functions by including a parametric function application; represented abstractly by a function parameter and a sequence of terms.

$$\begin{array}{lcl} \mathit{Term} & ::= & \mathrm{ph}\langle\langle \mathbb{N}_1 \rangle\rangle \\ & | & \mathrm{var}\langle\langle \mathit{Var} \rangle\rangle \\ & | & \mathrm{fnapplic}\langle\langle \mathit{FunctionName} \times \mathrm{seq}\ \mathit{Term} \rangle\rangle \\ & | & \mathrm{termparam}\langle\langle \mathit{FunctionParam} \times \mathrm{seq}\ \mathit{Term} \rangle\rangle \end{array}$$

Example 4.1 The abstract term

$$\mathrm{fnApplic}(f, \langle \mathrm{var}\, x, \mathrm{termParam}(G, \langle \mathit{fnApplic}(c, \langle\, \rangle)\rangle)\rangle) \tag{1}$$

has the concrete representation

$$f(x, G(c)). \tag{2}$$

4.2 Instantiating parameters

Expressions are *instantiated* by replacing occurrences of parameters by other non-parametric expressions. Where all parameters in an expression are replaced the expression is said to have been *fully* instantiated, otherwise the expression is said to be *partially* instantiated.

To describe how parameters in an expression are to be replaced, a *formal parameter instantiation* is given. The instantiation is defined as a finite partial mapping from parameters to terms containing placeholders. The placeholders refer to the arguments of the parameter in the instantiation.

$$\mathit{Inst} == \mathit{FunctionParam} \mathrel{+\!\!\!\!\twoheadrightarrow} \mathit{Term}$$

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. An instantiation of a parameter F to a term t is represented by the notation $F \rightsquigarrow t$. The trivial instantiation ($\mathit{TrivInst}$) is defined as the unique instantiation that leaves all expressions unchanged after application. It is represented as an empty function.

Parameter instantiations are often defined using placeholders. The instantiation $F \rightsquigarrow ph(1) + ph(2)$ indicates that applications of F with at least two arguments are replaced by the application of $+$ to the first and second arguments of F. More concretely, $F(x, y)$ would be replaced by $x + y$ using this instantiation.

The function *instantiate* applies an instantiation to an expression to yield a new expression.

$$\left|\ \mathit{instantiate} : \mathit{Term} \times \mathit{Inst} \nrightarrow \mathit{Term}\right.$$

For terms, the *instantiate* function is defined by considering placeholders and variables, non-parametric function applications and parametric function applications separately. The *instantiate* function is defined informally for terms as follows:

- placeholders[2] and unbound variables are left unchanged;
- non-parametric function applications are instantiated by instantiating the argument list;
- suppose $F \leadsto e$ is part of the instantiation map, then a parametric function of the form $F(a_1, .., a_m)$ is instantiated by firstly replacing the function application by e, then replacing any occurrences of placeholders $ph(j)$ where $j \in 1 \mathrel{..} m$ in e by the term that results from instantiating a_j.

Example 4.2 Suppose F, G, H are parameters, and an instantiation i, of the above parameters, is defined as:

$$i == \{F \leadsto f(ph(2), g(ph(1), ph(2))), G \leadsto h(ph(1)), H \leadsto ph(2)\} \tag{3}$$

Then the term $t_1 == G(H(c, x + y))$ is instantiated by i as follows:

$$\begin{aligned} instantiate(t_1, i) &= instantiate(G(H(c, x + y)), i) && (4)\\ &= h(instantiate(H(c, x + y), i)) && (5)\\ &= h(x + y) && (6) \end{aligned}$$

The term $t_2 == F(H(d, x), G(x))$ is instantiated by i as follows:

$$\begin{aligned} instantiate(t_2, i) &= instantiate(F(H(d, x), G(x)), i) && (7)\\ &= f(\Delta_2, g(\Delta_1, \Delta_2)) && (8) \end{aligned}$$

where Δ_1 is the result of instantiating the first argument of the call to F, and Δ_2 is the result of instantiating the second argument of the call to F, i.e.:

$$\begin{aligned} &\Delta_1 = instantiate(H(d, x), i) = x && (9)\\ &\Delta_2 = instantiate(G(x), i) = h(x) && (10) \end{aligned}$$

Therefore the result of instantiation the term t_2 by i is:

$$instantiate(t_2, i) = f(h(x), g(x, h(x))). \tag{11}$$

[2] Normally terms which are to be instantiated would not contain placeholders; however this case is included for completeness and will be useful later.

4.3 Matching terms

Two mathematical expressions p and q are said to match if there is an instantiation i such that instantiating p with i yields q. We refer to the expression p which possibly contains parameters as the *pattern*, while q is referred to as the *query*. The top level specification of *match* for terms is given by:

$$\begin{array}{|l}
match : \mathbb{P}(Term \times Term \times Inst) \\
\hline
\forall (p, q, i) \in match \bullet instantiate(p, i) = q
\end{array}$$

We now give an informal description of an algorithm for matching mathematical expressions that works on structural induction on the pattern. The algorithm for matching terms follows the structure of the abstract syntax for terms given in Section 4.1. Note that while placeholders only occur in expressions used in instantiations, they are considered here for completeness.

case 1:

$p = \mathrm{var}\, x$, where x is a variable.

p matches only the term x with a trivial instantiation.

case 2:

$p = ph(j)$, where j is a natural number > 0.

p matches queries $q = ph(j)$ with a trivial instantiation.

case 3:

$p = f(a_1, .., a_m)$, where f is a non-parametric function.

p matches queries of the form $f(b_1, .., b_m)$, where a_j matches b_j for each j. The set of matches can be formed by merging (where possible) the sets of instantiations formed by matching a_j against b_j for each j.

case 4:

$p = F(a_1, .., a_m)$, where F is a parametric function.

The set of matches of p with an arbitrary term e is formed by considering all disjoint sets of subtrees of e for which each subtree e_{sub} in the set matches an a_j for some $j \in 1 \mathinner{..} m$. The sets of instantiations formed by matching each e_{sub} in the disjoint set with the corresponding a_j are merged to form a set of instantiations. These are then merged with the instantiation $\{F \leadsto e'\}$, where e' is formed by replacing each subtree by the corresponding placeholder.

The complete set of instantiations is formed by taking the union of instantiation sets for all disjoint sets of subtrees, and finally filtering out any instantiations that contain free variables.

Example 4.3 Given a pattern $p = F(G(x, y), H(x, y))$ where F, G and H are all parameters, together with a query $q = (x + y) * (y - x)$. Two (of a number of) possible ways of matching p against q are:

$$\{F \rightsquigarrow ph(1) * ph(2), G \rightsquigarrow ph(1) + ph(2), H \rightsquigarrow ph(2) - ph(1)\} \tag{12}$$

$$\{F \rightsquigarrow ph(2) * ph(1), G \rightsquigarrow ph(2) - ph(1), H \rightsquigarrow ph(1) + ph(2)\} \tag{13}$$

The first match is generated by considering the following disjoint subtrees of q:

$$\{x + y, y - x\} \tag{14}$$

The first subtree $x + y$ is matched against the first argument of the function application F, i.e. $G(x, y)$, and the second subtree $y - x$ is matched against the second argument of F, i.e. $H(x, y)$. Replacing the subtrees $x+y$ and $y-x$ with the placeholders $ph(1)$ and $ph(2)$ respectively, results in the instantiation $F \rightsquigarrow ph(1) * ph(2)$.

By following the same process it can be shown that $G(x, y)$ matches $x + y$ with instantiation $G \rightsquigarrow ph(1) + ph(2)$. Similarly $H(x, y)$ matches $y - x$ with instantiation $H \rightsquigarrow ph(2) - ph(1)$.

The second match that is listed above can be generated in much the same way. Like the first match, the subtrees $x + y$ and $y - x$ are considered, but this time the first subtree is matched against the second argument of F, and the second subtree is matched against the first argument of F. □

5 Associative commutative matching

In this section we describe a relaxed form of matching, referred to as *associative commutative matching*, or AC-matching, which uses the properties of associative commutative operators to achieve more matches.

5.1 Associative commutative operators

Associative and commutative operators are special classes of operators defined as follows:

Definition 5.1 An operator f is said to be *associative* if for all u, v, w

$$f(f(u, v), w) = f(u, (f(v, w))$$

Example 5.2 Associative operators include the functions $+$, $*$ and $-$ over the set of integers. The binary connectives $\wedge$, $\vee$, $\Rightarrow$ and $\Leftrightarrow$ are also associative.

Definition 5.3 An operator f is said to be *commutative* if for all u, v

$$f(u, v) = f(v, u).$$

Example 5.4 The functions $+$ and $*$, as well as the binary connectives $\wedge$, $\vee$ and $\Leftrightarrow$ are all commutative. The function $-$ and the binary connective $\Rightarrow$ are examples of operators that are associative but not commutative.

Definition 5.5 Two expressions e_1 and e_2 are said to be AC-equivalent, written $e_1 =_{AC} e_2$ iff the two expressions can be proved equal using only the laws of associativity and commutativity for any AC operations in the two expressions, and renaming of bound variables.

It is straightforward to show that the relation $=_{AC}$ is an equivalence relation.

Example 5.6 Assuming that the operator f is AC, then the expressions $f(f(a, g(b)), c)$ and $f(f(g(b), c), a)$ are AC equivalent. This can be proved using the properties of associativity and commutativity as follows:

$$\begin{aligned} f(f(a, g(b)), c) &=_{AC} f(f(g(b), a), c) && \text{commutativity} && (15)\\ &=_{AC} f(g(b), f(a, c)) && \text{associativity} && (16)\\ &=_{AC} f(g(b), f(c, a)) && \text{commutativity} && (17)\\ &=_{AC} f(f(g(b), c), a) && \text{associativity} && (18) \end{aligned}$$

5.2 Flattening expressions

Expressions can be *flattened*, with respect to a given AC operator, by modifying the abstract syntax to allow two or more branches in the syntax tree below the operator (so in effect "removing the bracketing"). For example, assuming that $+$ is AC, the term $((a + b) + c)$ can be flattened with respect to $+$, to give $a + b + c$. The function *flatten* takes an AC-function and a term, and flattens the term with respect to the AC-function.

$$\mid \ \mathit{flatten} : \mathit{FunctionName} \times \mathit{Term} \rightarrow \mathrm{seq}\ \mathit{Term}$$

Note that if the outermost term is not the AC-operator in question, then no flattening will be done. For example the term $g((a + b) + c)$ will remain unchanged when flattened with respect to $+$.

5.3 Specification of AC-matching

A pattern p is said to match a query q up to AC-equivalence, if there exists an instantiation i, such that the result of instantiating p under i is AC-equivalent to q, i.e.:

$$\begin{array}{|l}
matches_{AC} : \mathbb{P}(Term \times Term \times Inst) \\
\hline
\forall\, p, q : Term;\ i : Inst \bullet \\
\qquad matches_{AC}(p, q, i) \Leftrightarrow instantiate(p, i) =_{AC} q
\end{array}$$

Example 5.7 Suppose that f is an AC operator, and that G is a parametric function. Then the pattern,

$$p == f(f(G(x, y), z), h(c)) \tag{19}$$

is equivalent to the query

$$q == f(f(x, h(c)), f(z, y)) \tag{20}$$

under the instantiation

$$i = \{G \leadsto f(ph(1), ph(2))\} \tag{21}$$

To show this is true, we firstly instantiate p with respect to i, then use the laws of associativity and commutativity to transform the instantiated pattern into the query, i.e.:

$$\begin{aligned}
instantiate(p, i) &= f(f(f(x, y), z), h(c)) && (22)\\
&=_{AC} f(f(x, y), f(z, h(c))) && (23)\\
&=_{AC} f(f(x, y), f(h(c), z)) && (24)\\
&=_{AC} f(f(x, f(y, h(c))), z)) && (25)\\
&=_{AC} f(f(x, f(h(c), y), z)) && (26)\\
&=_{AC} f(f(x, h(c)), f(y, z)) && (27)\\
&=_{AC} f(f(x, h(c)), f(z, y)) && (28)
\end{aligned}$$

6 An algorithm for AC-matching

The match algorithm can be extended by considering AC operators, and matching up to AC-equivalence. The $match_{AC}$ function extends $match$ for AC-functions.

$$\begin{array}{|l}
match_{AC} : Term \times Term \to \mathbb{F}\, Inst
\end{array}$$

For terms, the function $match_{AC}$ is implemented in a similar manner to $match$ (see section 4.3), except two extra cases are considered.

6.1 *Matching non-parametric AC-functions*

We begin by extending the matching algorithm by considering the case where the pattern and query are both non-parametric function applications of the same associative commutative functions (the pattern may include parameters in its subterms). An example is matching the pattern $(G(x,y)+z)+h(c)$ and the query $(x+h(c))+(z+y)$. One possible match can be achieved by instantiating $G \rightsquigarrow ph(1)+ph(2)$. We describe how this match is generated after describing the algorithm.

We assume the pattern p is of the form $p = f(a_1,\ldots,a_m)$ and the query q is of the form $q = f(b_1,\ldots,b_n)$, where f is a non-parametric, AC-function. In this case the algorithm for matching p against q proceeds as follows:

Step 1:

Apply the function *flatten* to the argument sequences $a_1,\ldots,a_m$ and $b_1,\ldots,b_n$ of the pattern and query respectively, to get the sequences $a'_1,\ldots,a'_j$ and $b'_1,\ldots,b'_k$, where $j \geq m, k \geq n$;

Step 2:

Remove any duplicates from these sequences, i.e. any identical terms that appear in both sequences, to give the sequences α and β respectively;

Step 3:

Generate the set of partitions of the sequence β, into a sequence of bags Φ such that:

(a) $\#\Phi = \#\alpha$

(b) $\sum_{k=1}^{\#\Phi} \mathit{count}\,\Phi(k) = \#\beta$

(c) for each $k \in 1\,..\,\#\alpha$

$$\mathit{count}\,\Phi(k) = \begin{cases} \geq 1 \text{ if } \alpha(k) \text{ is a parametric function application} \\ 1 \quad \text{otherwise} \end{cases}$$

(d) for each $k \in 1\,..\,\#\beta$ there exists $j \in 1\,..\,\#\Phi$ such that $\beta(k)\,\mathrm{in}\,\Phi(j)$

(e) each element appears the same number of times in the sequence of bags Φ that it appeared in the sequence β

Step 4:

For each partition sequence Φ generated in the previous step, form the sequence of terms γ by mapping each bag to a term as follows:

(a) for bags with a single element of the form $[\![u]\!]$ return the term u

(b) for bags $[\![u_1, \ldots, u_v]\!]$ with more than one element return the term $f(u_1, \ldots, u_v)$

Step 5:

Apply the function $match_{AC}(\alpha, \gamma)$ to get a set of matches for each sequence of terms γ generated in step 4.

Example 6.1 Given the pattern $p = (G(x, y) + z) + h(c)$ where $+$ is an AC-function and G is a parametric function, and query $q = (x + h(c)) + (z + y)$ from Ex. 5.7. Then the algorithm for generating matches for p and q proceeds as follows.

(i) flatten p and q with respect to the AC-operator $+$ to give the sequences:

$$\langle G(x, y), z, h(c)\rangle$$
$$\langle x, h(c), z, y\rangle$$

(ii) Remove elements which appear in both sequences to give the sequences:

$$\alpha = \langle G(x, y)\rangle$$
$$\beta = \langle x, y\rangle$$

(iii) Partition β to give the sequence of bags $\langle [\![x, y]\!]\rangle$

(iv) Form the sequence of terms $\gamma = \langle x + y\rangle$

(v) Generate the set of matches for $match_{AC}(\langle G(x, y)\rangle, \langle x + y\rangle)$. The result consists of the single instantiation $G \rightsquigarrow ph(1) + ph(2)$.

6.2 Matching higher-order functions against AC queries

The second extension that we consider is for queries of the form $g(b_1, .., b_m)$, where g is an AC function, and patterns of the form $F(as)$, where F is a parameter, and as is a argument list of the form $a_1, .., a_n$, and at least one of the arguments is a function application involving the AC function g.

A simple example of this is a pattern $G(x+z,y)$ matched against a query $y+x+z$. This can be achieved by instantiating $G \rightsquigarrow ph(1)+ph(2)$. A more complex example is for a pattern $G(x+y,y+x)$ matched against the query $x+y+y+x$. This can be achieved in under the following instantiations:

$$G \rightsquigarrow ph(1)+ph(1) \tag{29}$$
$$G \rightsquigarrow ph(1)+ph(2) \tag{30}$$
$$G \rightsquigarrow ph(2)+ph(2) \tag{31}$$

Notice that we do not include the instantiation $G \rightsquigarrow ph(2)+ph(1)$, since this is equivalent to the second instantiation listed above.

Before giving a sketch of the algorithm, we define the notion of the weight of a term with respect to a function f.

Definition 6.2 The *weight* of a term t with respect to a function f is defined as follows:

$$weight(t,f) = \begin{cases} \#\mathit{flatten}(t,f) & \text{if } f \text{ is an AC function} \\ 1 & \text{otherwise} \end{cases} \tag{32}$$

Example 6.3 For an AC function $+$, the weight of $x+(G(y)+f(c))$ is 3, the weight of x is 1, the weight of $f(x+y,x+z)$ is 1, and the weight of $G(x+z)$ is 1.

For queries of the form $g(b_1,..,b_m)$, where g is an AC function (it is assumed that the query has been flattened with respect to g), and patterns of the form $F(as)$, where F is a parameter, and as is a argument list of the form $a_1,..,a_n$, the algorithm proceeds as follows:

Step 1:

Generate the set of multisets, with k elements, of the form $[\![u_1,..,u_k]\!]$, where each u_i is a natural numbers. Each multiset should satisfy the following conditions:

(i) each u_i represents an index (a natural number, $\mathbb{N}$) to the arguments of the pattern, such that for each $i \in 1\,..\,k$,

$$u_i \in 1\,..\,n$$

(ii) the bags are non-empty, such that the sum of the corresponding pattern arguments' weight with respect to g is less than or equal to the number of arguments of the flattened query, i.e.,

$$1 \leq \sum_{j=1}^{k} weight(as(u_j), g) \leq m.$$

Step 2:

For each bag, generate the set of matches by considering bags with a single element and those with multiple elements separately, as follows:

(i) for bags of the form $[\![u]\!]$, generate the set of matches between the expression $as(u)$, and the query $g(b_1, .., b_m)$, and merge each result with the instantiation $F \leadsto ph(u)$.

(ii) for bags of the form $[\![u_1, .., u_k]\!]$, where $k > 1$, generate the set of matches between the expression $g(as(u_1), .., as(u_k))$, and the query $g(b_1, .., b_m)$, merging each result with the instantiation $F \leadsto g(ph(u_1), .., ph(u_k))$.

Example 6.4 To illustrate the algorithm, we consider matching the pattern $G(x + y, x + z, y, z)$, where G is a parameter, against the query $x + y + z$, where we assume $+$ is an AC operator. We begin by noting that the query has been flattened with respect to $+$. The first step then is to generate the set of multisets satisfying the conditions from Step 1 above. We note that the weight with respect to $+$ of the first and second arguments of the pattern is 2, while the weight of the other two arguments is 1. Therefore the set of possible multisets is:

$$\begin{aligned}\{&[\![1]\!], [\![2]\!], [\![3]\!], [\![4]\!],\\ &[\![1,3]\!], [\![1,4]\!], [\![2,3]\!], [\![2,4]\!], [\![3,3]\!], [\![3,4]\!], [\![4,4]\!],\\ &[\![3,3,3]\!], [\![3,3,4]\!], [\![3,4,4]\!], [\![4,4,4]\!]\}\end{aligned} \tag{33}$$

For each of these multisets we form a new term, as described in step 2 of the algorithm. For example, for the multiset $[\![1,3]\!]$, the term $(x + y) + y$ is formed, consisting of one instance of the first argument of the pattern, and one instance of the third argument. For each multiset an instantiation of the parameter G is also created; for the multiset $[\![1,3]\!]$ the instantiation $G \leadsto ph(1) + ph(3)$ is formed. Each new term is then matched against the query and any instantiations representing successful matches are merged with the corresponding new created instantiation of G.

In this example there are two successful matches $G \leadsto ph(1) + ph(4)$ and $G \leadsto ph(2) + ph(3)$, derived from the multisets $[\![1,4]\!]$ and $[\![2,3]\!]$ respectively. In the first case the term $(x + y) + z$ is created, which matches the query with the trivial instantiation. This is merged with the instantiation $G \leadsto ph(1) + ph(4)$

to give the first answer. In the second case the term $(x + z) + y$ is created, which again matches the query with the trivially.

7 Discussion

7.1 Incompleteness

The algorithm described in this paper is by no means complete in the sense that for some queries and patterns there are instantiations (beyond those that are in some sense equivalent to one that is returned) satisfying the specification that are not returned. Indeed it is simple to give an example satisfying the specification that is not returned by the algorithm.

Example 7.1 Consider the pattern $F(x + c, y, u, v)$ where F is a parameter, and the query $x + (u * v) + y + c$. By instantiating $F \rightsquigarrow ph(1) + ph(2) + (ph(3) * ph(4))$ we get the term $x + c + y + (u * v)$, which can readily be seen to be AC equivalent to the query. However this instantiation is not generated by our algorithm.

The incompleteness of the algorithm will probably not appeal to the purists, however it must be remembered that our aim was to increase level of recall associated with matching, while not adversely affecting the level of efficiency and automation. In this sense our algorithm achieves the stated goals.

There does not appear to be any published decidability or complexity results for higher-order AC matching, however several results have been published for first order AC matching [7,3]. These papers prove that first order AC matching is decidable, however there are pathological cases which take exponential time to compute. Eker [3], shows that for cases where there is only a single AC function, solutions can be calculated in polynomial time.

7.2 Building retrieval tool support

The algorithm described in this paper has been used as part of the CARE library retrieval tool [5]. The algorithm (as with other parts of the retrieval tool), has been implemented in Prolog. Because Prolog is a high-level language, the implemented algorithm is quite similar to the algorithm described in this paper. The algorithm uses backtracking to calculate all solutions.

The retrieval tool is based on matching fragment specifications, corresponding to user requirements (the query) and library components (the pattern). The fragment specifications are represented using pre- and post-conditions, and matching is done by matching the corresponding pre- and post-conditions, and then merging the resulting instantiations (if there are any).

This represents fine-grained matching, however library components in CARE (referred to as templates), typically consist of multiple fragments. So matching is extended further to include matching multiple user requirements against library templates. Individual fragments are matched (as described above), then the resulting instantiations are merged to give a complete match.

A further enhancement that could be made to the retrieval tool is to combine theorem prover based matching with our existing pattern based matching. This could be achieved in two ways. The first approach would be to implement separate matching algorithms, i.e., theorem prover based and pattern matching based, then make a decision on which one to use based on the form of the pattern and query. An alternative is to use a theorem prover that supports higher-order logic [14,4]. Such provers provide support for higher-order unification, however they could be enhanced by adding support for associative commutative matching, as described in this paper.

8 Conclusions

In this paper we have describe a higher-order associative commutative pattern matching algorithm. While the algorithm is not complete (in that it does not return all solutions satisfying the specification), it does meet our required goals of increasing the recall of syntactic-based pattern matching, while not overly compromising the efficiency, and maintaining automation. We have briefly described how the algorithm is used as part of a component retrieval tool, and discussed how further enhancements could be achieved by combining AC matching with semantic-based matching techniques, or how it might enhance the capabilities of theorem provers based on higher-order logics.

Acknowledgements

This work was funded by Australian Research Council Discovery Grant DP0208046, Compilation of Specifications.

References

[1] Abrial, J-R., *The B Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[2] Dowek, G., Third order matching is decidable. *Annals of Pure and Applied Logic*, 69:135–155, 1994.

[3] Eker, S. M., Improving the efficiency of AC matching and unification. Technical Report RR-2104, INRIA Lorraine, 1993.

[4] Gordon, M. J. C. and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[5] Hemer, D. and P. Lindsay, Supporting component-based reuse in CARE. In Michael Oudshoorn, editor, *Proceedings of the Twenty-Fifth Australasian Computer Science Conference*, volume 4 of *Conferences in Research and Practice in Information Technology*, pages 95–104. Australian Computer Society Inc., January 2002.

[6] Hemer, D. 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. SVRC TR 95-52.

[7] Hermann, M. and P. G. Kolaitis, Computational complexity of simultaneous elementary matching problems. *Journal of Automated Reasoning*, 23:107–136, 1999.

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

[9] Jeng, J-J. and B. H. C Cheng, Specification matching for software reuse: A foundation. In *Proc. of ACM Symposium on Software Reuse*, pages 97–105, April 1995.

[10] Lincoln, P. and J. Christian, Adventures in associative-commutative unification. In Claude Kirchner, editor, *Unification*, pages 393–416. Academic Press, 1990.

[11] McIlroy, M. D., Mass produced software components. *Software Engineering Concepts and Techniques*, pages 88–98, 1969.

[12] Miller, D., Unification of simply typed lambda-terms as logic programming. In P.K. Furukawa, editor, *Proc. 1991 Joint Int. Conf. Logic Programming*, pages 253–281. MIT Press, 1991.

[13] Nipkow, T., Functional unification of higher-order patterns. In *Proceedings of the 8th IEEE Symposium Logic in Computer Science*, pages 64–74. IEEE Computer Society Press, June 1993.

[14] Paulson, L. C., Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[15] Perry, D. E. 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.

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

[17] Schmidt, H. W. and W. Zimmermann, A complexity calculus for object-oriented programs. *Journal of Object-Oriented Systems*, 1(2):117–147, 1994.

[18] Schumann, Johann M., *Automated Theorem Proving in Software Engineering.* Springer Verlag, 2001.

[19] Stickel, M., 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.

[20] Williamson, K. and M. Healy, Industrial application of software synthesis via category theory – case studies using specware. *Automated Software Engineering*, 8:7–30, 2001.

[21] Wolfram, D. A., The decidability of higher-order matching. In Jorg Siekmann Franz Baader and Wayne Snyder, editors, *Proceedings of the Sixth International Workshop on Unification, UNIF'92*, 1992.

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