

Chapter 7

Matching Units

7.1 Introduction

In Chapter 5 an approach to retrieving formally specified components was proposed. The approach is based on matching the user's requirements expressed by a formal specification (the query) against the specifications of library components (patterns). The approach is integrated with component adaptation by describing matches between a query and a pattern in terms of an adaptation of the pattern.

In this chapter matching is developed for units in CARE. This chapter assumes that an approach to unit adaptation has been developed (indeed this chapter builds on the framework for adapting units in CARE given in Section 4.3).

Units in the context of this thesis can be thought of as compound constructs, built from a number of smaller, simpler subunits. A pattern matches a query if there is an adaptation of the pattern which is *structurally equivalent* to the query. Recall that two units are *structurally equivalent* if their corresponding subparts are equivalent (Chapter 6 considered a number of different equivalence relations for expressions). Having defined unit matching in this sense, algorithms for finding the matches between a query and pattern can be developed.

The first step in developing the unit matching algorithms involves identifying the basic subunits and developing algorithms for matching these subunits. For example, in CARE fragment specifications the subunits are expressions, variable declarations

and identifiers.

Next algorithms for matching these units can be implemented by matching the corresponding subunits (e.g. matching the pre-conditions of two fragment specifications). Each of the matches between subunits is described in terms of an adaptation. If these individual adaptations are not inconsistent, then they can be merged to form a single adaptation, representing a match between the complete units.

While the exact details of unit matching algorithms differ for the various kinds of units in CARE, they are all based on matching up to structural equivalence. Indeed this approach to matching is general enough to be applicable to a much wider range of applications and languages.

The unit matching algorithms developed for CARE in this chapter are based on the unit adaptation techniques described in Chapter 4: i.e. parameter instantiation, identifier renaming, renaming of input and output arguments and reordering of arguments.

A simpler subset of the overall solution is developed firstly in Section 7.2, where only parameter instantiation and identifier renaming are considered. This section begins by discussing matching techniques for the subunits of CARE specifications (i.e. expressions, variable declarations and identifiers). Next the algorithms for matching the various units - types, fragments (simple and branching), operator declarations and assertions - are developed.

In Section 7.3 the matching algorithms (in particular those for fragments) are extended by considering the two other adaptation techniques: renaming of variables and reordering of variables. A discussion of how the approach to matching units can be applied in a wider context is given in Section 7.4.

The unit matching algorithms which follow are largely independent of the nature and implementation of the functions for matching mathematical expressions. The current prototype tool in CARE uses the matching algorithms defined in Chapter 6 but the framework easily allows others to be substituted.

7.2 A basic approach to unit matching

In this section a partial solution to unit matching is developed, where only parameter instantiation and unit-identifier renaming are considered. The algorithms are straightforward, but are given in full detail as a firm foundation for what follows.

The remainder of this section describes algorithms for matching the various sub-units which appear in CARE units. Then algorithms are given for matching the different kinds of units: simple fragments; branching fragments; types; operator declarations; and assertions. The completeness of each of these algorithms is considered.

Before defining matching in CARE, queries for each of kinds of CARE unit must be defined. The query for simple fragment matching consists of a name and a specification.

$$SFragQuery == FragmentName \times SimpleFragmentSpec$$

A branching fragment query consists of a name and a specification, i.e. it is similar to an unimplemented fragment.

$$BFragQuery == FragmentName \times BFragmentSpec$$

A type query is equivalent to a type specification.

$$TypeQuery == TypeName \times Set$$

7.2.1 Preliminary definitions

Two identifier renamings are mergeable if they agree for common domain values. Renamings are merged by applying function override.

$$\left| \begin{array}{l} \hline areMergeableRenamings : Renaming \times Renaming \rightarrow Renaming \\ \hline dom\ areMergeableRenamings = \\ \quad \{r_1, r_2 : Renaming \mid r_1 \triangleleft (dom\ r_1 \cap dom\ r_2) = r_2 \triangleleft (dom\ r_1 \cap dom\ r_2)\} \\ \quad \forall r_1, r_2 : Renaming \bullet areMergeableRenamings(r_1, r_2) = r_1 \oplus r_2 \end{array} \right.$$

7.2.2 Mathematical expressions

The first step is to develop algorithms for matching the mathematical expressions which appear in unit specifications, i.e. terms, formulae and sets (cf. Section 2.2). A relation *matches* is given which defines the meaning of expression matching. The matching algorithms defined in Chapter 6 could be reused here, however the algorithms developed in this chapter are independent of the expression matching algorithms, provided the following are given:

- a relation *matches* defining the meaning of expression matching
- a function *match* which returns the set of matches
- some notion of equivalence between expressions.

$$\left| \begin{array}{l} \text{matches} : \mathbb{P}(Fmla \times Fmla \times Inst) \\ \text{match} : Fmla \times Fmla \rightarrow \mathbb{F} Inst \end{array} \right.$$

Similar functions are required for terms and sets, as well as functions for unifying operator signatures. As with the earlier convention, for the *matches* and *match*, the pattern appears first in the argument list, while the query appears second.

7.2.3 Variable declarations

The input and output arguments of fragments are defined in terms of variable declarations (see *VarDecls* defined on page 32); an algorithm for matching variable declarations is therefore required. Recall that variable declarations declare the CARE types of variables; represented by pairs of variables and CARE types. In this section, where only parameter instantiation and identifier renaming are considered, two variable declarations match if they are the same up to renaming of type identifiers.

$$\left| \begin{array}{l} \text{matches}_{vs} : \mathbb{P}(VarDecls \times VarDecls \times Renaming) \\ \hline \forall vs_1, vs_2 : VarDecls; r : Renaming \bullet \\ \text{matches}_{vs}(vs_1, vs_2, r) \Leftrightarrow \text{rename}(vs_1, r) = vs_2 \end{array} \right.$$

This definition will be extended in Section 7.3, where renaming of variables, and reordering of the variable declaration will also be considered. The function *match_{vs}*

returns the set of matches between two variable declarations, described in terms of an identifier renaming. Note that if a match exists, it is unique, however sets of matches are used for consistency with what follows.

$$\mid \text{match}_{vs} : \text{VarDecls} \times \text{VarDecls} \rightarrow \mathbb{F} \text{Renaming}$$

Algorithm 9 contains an implementation for match_{vs} , which employs an accumulation like strategy. The algorithm begins by checking that the two lists are of equal length. If so, then variable and type pairs in corresponding positions in the lists are matched in turn, beginning with the first element in each list and progressively moving through to the last. At each stage a new renaming is accumulated; if at any stage the renamings are not consistent, then the algorithm terminates.

Algorithm 9 Calculate $rs = \text{match}_{vs}(vs_1, vs_2)$

```

1: if #vs1 = #vs2 then
2:   r := ∅
3:   while vs1 ≠ ⟨⟩ do
4:     (v1, t1) := head vs1
5:     (v2, t2) := head vs2
6:     if v1 = v2 ∧ t1 ∉ dom r then
7:       r := {t1 ↦ t2} ∪ r
8:     else if v1 = v2 ∧ r(t1) = t2 then
9:       {do nothing}
10:    else
11:      return ∅
12:    vs1 := tail vs1
13:    vs2 := tail vs2

```

Proposition 7.1 For each pattern vs_1 , query vs_2 and renaming r

$$r \in \text{match}_{vs}(vs_1, vs_2) \Rightarrow \text{matches}(vs_1, vs_2, r)$$

Proof: Suppose r is a renaming in $\text{match}_{vs}(vs_1, vs_2)$, it is necessary to show $\text{rename}(vs_1, r) = vs_2$. From line 1, $\#vs_1 = \#vs_2$ and from lines 3-11, for each pair (v_1, t_1) in vs_1 and

(v_2, t_2) in the corresponding position in vs_2 , $v_1 = v_2$ and $r(t_1) = t_2$. It follows from the definition of *rename* that $rename(vs_1, r) = vs_2$. \square

Completeness of the algorithm can be proved by observing that for each corresponding variable/type pair in the pattern and query, there is at most one way of renaming the type in the pattern. That is renaming the pattern type to the query type. This is handled on line 7 of the algorithm.

7.2.4 Simple fragments

Recall that *simple fragments* are a particular class of fragments which contain no branching in their specification. The subunits of a simple fragment are its identifier, input variable declarations, precondition, output variable declaration and postcondition. The corresponding subunit adaptations considered here are instantiation and renaming.

7.2.4.1 Matching simple fragments

A simple fragment matches a simple fragment query if the corresponding subunits match under some instantiation i and renaming r .

$$\begin{array}{|l} \hline matches_{sfrag} : \mathbb{P}(SimpleFragment \times SFragQuery \times Inst \times Renaming) \\ \hline \forall p : SimpleFragment; q : SFragQuery; i : Inst; r : Renaming \bullet \\ matches_{frag}(p, q, i, r) \Leftrightarrow \\ matches_{vs}(p.spec.invars, \pi_2(q).invars, r) \wedge \\ matches(p.spec.precond, \pi_2(q).precond, i) \wedge \\ matches_{vs}(p.spec.outvars, \pi_2(q).outvars, r) \wedge \\ matches(p.spec.postcond, \pi_2(q).postcond, i) \\ \hline \end{array}$$

Example 7.1 Given the simple fragment query q and pattern p in Fig. 7.1, where P and Q are parameters. Then p matches q under the following instantiation and renaming:

$$\begin{array}{l} \{P(a, b) \rightsquigarrow a \neq g(b), Q(a, b, c) \rightsquigarrow c \geq a + b\} \\ \{X \mapsto U, Y \mapsto V, Z \mapsto W\}. \end{array}$$

Query:

Fragment $q(u:U, v:V)$ has specification:

precondition $u \neq g(v)$

output w : W such that $w \geq u + v$.

Pattern:

Fragment $p(u:X, v:Y)$ has specification:

precondition $P(u, v)$

output w : Z such that $Q(u, v, w)$.

Figure 7.1: A simple fragment query and matching candidate

7.2.4.2 An algorithm for matching simple fragments

The function $match_{sfrag}$ returns the set of matches between a simple fragment query and a simple fragment. The matches are represented as pairs of instantiations and identifier renamings.

$$\mid match_{sfrag} : SimpleFragment \times SFragQuery \rightarrow \mathbb{F}(Inst \times Renaming)$$

Algorithm 10 contains an implementation for $match_{sfrag}$. Firstly the input and output declarations are matched, giving a set of renamings in each case; these sets are merged to form an overall set of identifier renamings. If this set is non-empty, then the preconditions and post-conditions are matched, using the function for matching formulae. A set of instantiations are returned in each case, which are then merged to form the overall instantiation set. The final result is formed by considering all ordered pairs formed by taking an instantiation and renaming from the two generated sets.

Proposition 7.2 *For a simple fragment pattern p and query q , instantiation i and renaming r :*

$$(i, r) \in match_{sfrag}(p, q) \Rightarrow matches_{sfrag}(p, q, i, r)$$

Proof: Suppose $(i, r) \in match_{sfrag}(p, q)$. From lines 4-6 there is a renaming r_1 such that $matches_{vs}(p.spec.invars, \pi_2(q).invars, r_1)$. From the implementation of $match_{vs}$, it can be argued that all identifiers in the input list of the pattern are included in the renaming r_1 . Furthermore, from the definition of *areMergeableRenamings* it follows

Algorithm 10 Calculate $as = match_{sfrag}(p, q)$

```

1:  $as := \emptyset$ 
2:  $is := \emptyset$ 
3:  $rs := \emptyset$ 
4:  $spec1 := p.spec$ 
5:  $spec2 := \pi_2(q)$ 
6:  $rs1 := match_{vs}(spec1.invars, spec2.invars)$ 
7:  $rs2 := match_{vs}(spec1.outvars, spec2.outvars)$ 
8: for  $r1 \in rs1, r2 \in rs2$  do
9:   if  $areMergeableRenamings(r1, r2, r)$  then
10:      $rs := \{r\} \cup rs$ 
11: if  $rs \neq \emptyset$  then
12:    $is1 := match(spec1.precond, spec2.precond)$ 
13:    $is2 := match(spec1.postcond, spec2.postcond)$ 
14:   for  $i1 \in is1, i2 \in is2$  do
15:     if  $areMergeableInsts(i1, i2, i)$  then
16:        $is := \{i\} \cup is$ 
17:   for  $r \in rs, i \in is$  do
18:      $as := \{(i, r)\} \cup as$ 

```

that all of the identifiers in the input list of the pattern are renamed to the same value by r . Therefore it follows that:

$$matches_{vs}(p.spec.invars, \pi_2(q).invars, r) \quad (7.1)$$

Using a similar argument, starting initially with lines 4,5 and 7 of Alg. 10, it can be shown that:

$$matches_{vs}(p.spec.outvars, \pi_2(q).outvars, r) \quad (7.2)$$

By lines 12,14-16 of the algorithm, and from Lemma 6.1, page 132, it follows that:

$$matches(p.spec.precond, \pi_2(q).precond, i) \quad (7.3)$$

Similarly, from lines 13-16 of Alg. 10 it follows that:

$$\text{matches}(p.\text{spec}.\text{postcond}, \pi_2(q).\text{postcond}, i) \quad (7.4)$$

The proof is completed by combining equations (7.1)-(7.4). \square

Note that this algorithm is clearly complete provided that the algorithm for matching expressions is complete.

7.2.5 Branching fragments

The class of fragments containing branching in their specifications, is now considered (see Section 2.3.3 for more details). The subunits of a branching fragment are its identifier, input variable declaration, precondition, guards, output variable declarations and postconditions. The corresponding subunit adaptations considered here are parameter instantiation and identifier renaming.

7.2.5.1 Matching branching fragments

A branching fragment pattern matches a query under a renaming r and an instantiation i , if the number of branches in the specifications of pattern and query are the same; and the corresponding input variables, preconditions, and branches match under i and r .

$$\begin{array}{|l} \hline \text{matches}_{\text{frag}} : \mathbb{P}(\text{BranchingFragment} \times \text{BFragQuery} \times \text{Inst} \times \text{Renaming}) \\ \hline \forall p : \text{BranchingFragment}; q : \text{BFragQuery}; i : \text{Inst}; r : \text{Renaming} \bullet \\ \text{matches}_{\text{bfrag}}(p, q, i, r) \Leftrightarrow \\ \text{matches}_{\text{vs}}(p.\text{spec}.\text{invars}, \pi_2(q).\text{invars}, r) \wedge \\ \text{matches}(p.\text{spec}.\text{precond}, \pi_2(q).\text{precond}, i) \wedge \\ \# \pi_2(q).\text{branches} = \# p.\text{spec}.\text{branches} \\ \forall k : 1 \dots \# \pi_2(q).\text{branches} \bullet \\ \text{matches}_{\text{sb}}((p.\text{spec}.\text{branches})(k), (\pi_2(q).\text{branches})(k), i, r) \end{array}$$

Two specification branches match under instantiation i and renaming r , if the guards and postcondition match under i , the output variables match under r and the reports match under r

$$\begin{array}{|l}
\hline
\text{matches}_{sb} : \mathbb{P}(\text{SpecBranch} \times \text{SpecBranch} \times \text{Inst} \times \text{Renaming}) \\
\hline
\forall sb_1, sb_2 : \text{SpecBranch}; i : \text{Inst}; r : \text{Renaming}; \\
\text{matches}_{sb}(sb_1, sb_2, i, r) \Leftrightarrow \\
\text{matches}(sb_1.\text{guard}, sb_2.\text{guard}, i) \wedge \\
\text{rename}(sb_1.\text{report}, r) = sb_2.\text{report} \wedge \\
\text{matches}_{vs}(sb_1.\text{outputvars}, sb_2.\text{outputvars}, r) \wedge \\
\text{matches}(sb_1.\text{postcond}, sb_2.\text{postcond}, i) \wedge
\end{array}$$

7.2.5.2 An algorithm for matching branching fragments

The function match_{bfrag} returns the set of matches between a branching fragment query and pattern; the matches are represented in terms of instantiation/renaming pairs.

$$\left| \text{match}_{frag} : \text{BranchingFragment} \times \text{BFragQuery} \rightarrow \mathbb{F}(\text{Inst} \times \text{Renaming}) \right.$$

The algorithm for matching a fragment pattern p against a fragment query q proceeds as follows (described here informally for brevity):

1. match the input variables and types of q and p ;
2. match the output variables and types for each branch of q with the output variables in the corresponding branch of p (this only succeeds when there are an equal number of branches);
3. match the reports for each branch of q against the corresponding report in p ;
4. match the guard and postconditions for each branch of q with the guard and postconditions in the corresponding branch of p ;
5. finally, match the preconditions of q and p .

At each stage of the algorithm, zero or more pairs of instantiation and renamings are found. The results of these are merged with the pairs found in the previous stage using the merge functions.

The algorithm could be formalised in a similar manner to the algorithm given in the previous section for match_{sfrag} , with matching of guards added, and a loop added for matching of the branches. The proof of correctness and completeness of

such a formalised algorithm would follow the same arguments as given in the proof for $match_{sfrag}$.

7.2.6 Types

CARE type specifications consist simply of the type's name and a set expression representing its values.

Type adaptations consist of a renaming and an instantiation (see page 88 for more details):

$$TypeAdapt == Inst \times Renaming$$

A type matches a type query under an instantiation i and renaming r if the corresponding specification parts match under i and the type names are the same up to renaming under r .

$$\left| \begin{array}{l} matches_{type} : \mathbb{P}(Type \times TypeQuery \times TypeAdapt) \\ \hline \forall p : Type; q : TypeQuery; a : TypeAdapt \bullet \\ \quad matches_{type}(p, q, a) \Leftrightarrow \\ \quad \quad matches(p.spec, \pi_2 q, \pi_1 a) \\ \quad \quad \Rightarrow (p.name \mapsto \pi_1 q) \in \pi_2 a \end{array} \right.$$

The function $match_{type}$ returns the set of matches between a type query and type. An example which shows a pattern parameterised over the sets X and Y matched against a query is given in Fig.7.2.

$$\left| \quad match_{type} : Type \times TypeQuery \rightarrow \mathbb{F} TypeAdapt \right.$$

Pattern:

Type A has specification: $RelOf(X, Y)$.

Query:

Type B has specification: $RelOf(\mathbb{N}, ListsOf(\mathbb{N}))$.

Instantiation:

$\{X \rightsquigarrow \mathbb{N}, Y \rightsquigarrow ListsOf(\mathbb{N})\}$

Renaming:

$\{A \mapsto B\}$

Figure 7.2: Matching type specifications

An implementation for $match_{type}$ is given in Alg. 11; it involves firstly matching the type names, introducing a renaming where required, then matching the set expressions.

Algorithm 11 Calculate $as = match_{type}(p, q)$

```

1:  $as := \emptyset$ 
2:  $r := \{p.name \mapsto \pi_1(q)\}$ 
3:  $is := match(p.spec, \pi_2(q))$ 
4: for  $i \in is$  do
5:    $as := \{(r, i)\} \cup as$ 

```

Proposition 7.3 For a type pattern p and query q , and type adaptation a :

$$a \in match_{type}(p, q) \Rightarrow matches_{type}(p, q, a)$$

Proof: If $a \in match_{type}(p, q)$, then there exists r and i such that $a = (r, i)$, $r = \{p.name \mapsto \pi_1(q)\}$ and $i \in match(p.spec, \pi_2(q))$. From the assumption that $match$ is correct, it follows that $matches(p, q, a)$ is true. \square

Proof of completeness follows from the completeness of $match$.

7.2.7 Operator declarations

The query for matching operator declarations (see Section 2.3.1) is itself an operator declaration, however it is assumed that the query operator name is non-parametric. The signature of the query operator may however contain parameters. The subunits of an operator declaration are its identifier and signature. The corresponding subunit adaptation considered is instantiation.

A pattern matches a query under an instantiation i if the operator names match under i , and the signatures unify under i . Operator names match if either the pattern operator is a parameter, in which case the operator is instantiated to the query operator; or if the operators have the same name.

Two operator declarations match under an instantiation i if they are the same kind of operator and the signatures match under i .

Pattern: $f : \text{seq } \mathbb{N} \rightarrow A$ **Query:** $reverse : \text{seq } X \rightarrow \text{seq } X$ **Instantiation:** $X \rightsquigarrow \mathbb{N}, A \rightsquigarrow \text{seq } \mathbb{N}, f(a) \rightsquigarrow reverse(a)$

Figure 7.3: Matching operator declarations

$$\left| \begin{array}{l} matches_{op} : \mathbb{P}(OperatorDecl \times OperatorDecl \times Inst) \end{array} \right.$$

The function $match_{op}$ returns the set of instantiations representing matches between two operator declarations: Fig. 7.3 shows a match between the declaration of the parametric function f against the declaration of the function $reverse$. The signature of f is parameterised over the set A , while the signature of $reverse$ is parameterised over the set X .

$$\left| \begin{array}{l} match_{op} : OperatorDecl \times OperatorDecl \rightarrow \mathbb{F} Inst \end{array} \right.$$

7.2.8 Assertions

The subunits of an assertion in CARE are its identifier and statement. The subunit adaptation considered here is parameter instantiation.

Two assertions p and q match if their statements match under some instantiation i . Fig. 7.4 shows a match between an applicability condition from the Accumulator template (see page 271), and a theorem for the $reverse$ operator. The applicability condition (the pattern) is parameterised over the set E and the functions f and dh .

$$\left| \begin{array}{l} matches_{assert} : \mathbb{P}(Assertion \times Assertion \times Inst) \\ \hline \forall p, q : Assertion; i : Inst \bullet \\ matches_{assert}(p, q, i) \Leftrightarrow \\ matches(p.statement, q.statement, i) \end{array} \right.$$

A function for finding the set of matches between two assertions is defined as follows:

$$\left| \begin{array}{l} match_{assert} : Assertion \times Assertion \rightarrow \mathbb{F} Inst \\ \hline match_{assert}(p, q) = match(p.statement, q.statement) \end{array} \right.$$

Pattern:

$$\forall h : E; t : \text{seq } E \bullet f(\text{append}(h, t)) = dh(f(t), h)$$
Query:

$$\forall h : \mathbb{N}; t : \text{seq } \mathbb{N} \bullet \text{reverse}(\text{append}(h, t)) = \text{reverse}(t) \hat{\ } \langle h \rangle$$

Instantiation: $E \rightsquigarrow \mathbb{N}, f(a) \rightsquigarrow \text{reverse}(a), dh(a, b) \rightsquigarrow b \hat{\ } \langle a \rangle$

Figure 7.4: Matching assertions

Proof of correctness and completeness for $\text{match}_{\text{assert}}$ follows trivially from its definition (assuming match is correct and complete).

7.3 Extending the matching algorithm

In Section 7.2 a unit matching algorithm was developed which considered instantiation of parameters and renaming of identifiers only. However the framework for unit adaptation in CARE described in Section 4.3 also included renaming and reordering of unit arguments, in particular the input and output argument lists for fragments. In the following section the unit matching algorithm is extended by considering these two adaptation techniques.

7.3.1 Renaming of input and output variables

The names used for variables in the input and output declarations of fragments are not important, provided they are used consistently throughout the scope of the fragment. Therefore for matching purposes, the input and output variables of the pattern can be renamed to the corresponding query variable names. The matches_{vs} relation can be extended to include renaming of variables as follows:

$$\left| \begin{array}{l} \text{matches}_{vs} : \mathbb{P}(\text{VarDecls} \times \text{VarDecls} \times \text{Renaming} \times \text{VarRenaming}) \\ \forall vs_1, vs_2 : \text{VarDecls}; r : \text{Renaming}; vr : \text{VarRenaming} \bullet \\ \text{matches}_{vs}(vs_1, vs_2, r, vr) \Leftrightarrow \\ \quad \#vs_1 = \#vs_2 \wedge \\ \quad \forall k : 1 \dots \#vs_1 \bullet vr(\pi_1 vs_1(k)) = \pi_1 vs_2(k) \wedge \\ \quad r(\pi_2 vs_1(k)) = \pi_2 vs_2(k) \end{array} \right.$$

Example 7.2 The pattern $x:X, y:Y, z:Z$ matches the query $a:A, b:B, c:C$, under the identifier renaming r and variable renaming vr given as follows:

$$\begin{aligned} r &= \{X \mapsto A, Y \mapsto B, Z \mapsto C\} \\ vr &= \{x \mapsto a, y \mapsto b, z \mapsto c\} \end{aligned}$$

7.3.2 Reordering of input and output variables

Like the names given to fragment arguments, the order in which input and output variables appear in a fragment specification is not of great importance: e.g. whether one defines $\text{cons}(e:\text{Elem}, s:\text{List})$ or $\text{cons}(s:\text{List}, e:\text{Elem})$ is largely a matter of taste. So a further enhancement to unit matching is to make the matching function insensitive to the order of inputs and outputs of fragments. The definition of variable declaration matching can be extended to include reordering of variables in the pattern (together with renaming of identifiers and variables as given above).

$$\left| \begin{array}{l} \text{matches}_{vs}^* : \mathbb{P}(\text{VarDecls} \times \text{VarDecls} \times \text{Renaming} \\ \quad \times \text{VarRenaming} \times \text{Permutation}) \\ \hline \forall vs_1, vs_2 : \text{VarDecls}; r : \text{Renaming}; vr : \text{VarRenaming}; \\ \quad p : \text{Permutation} \bullet \\ \text{matches}_{vs}^*(vs_1, vs_2, r, vr, p) \Leftrightarrow \\ \quad \text{matches}_{vs}^*(\text{permute}(vs_1, p), vs_2, r, vr) \end{array} \right.$$

Example 7.3 The pattern $x:X, y:X, z:Z$ matches the query $a:A, b:B, c:A$ under variable renaming vr_1 , identifier renaming r_1 and permutation π_1 (which maps the first variable to the third position, the second variable to the first position and the third variable to the second position),

$$\begin{aligned} vr_1 &= \{x \mapsto c, y \mapsto a, z \mapsto b\} \\ r_1 &= \{X \mapsto A, Z \mapsto B\} \\ \pi_1 &= \{1 \mapsto 3, 2 \mapsto 1, 3 \mapsto 2\} \end{aligned}$$

or with variable renaming vr_2 , identifier renaming r_2 and permutation π_2

$$\begin{aligned} vr_2 &= \{x \mapsto a, y \mapsto c, z \mapsto b\} \\ r_2 &= \{X \mapsto A, Z \mapsto B\} \\ \pi_2 &= \{1 \mapsto 1, 2 \mapsto 3, 3 \mapsto 2\} \square \end{aligned}$$

7.3.3 Extending fragment matching

Fragment matching can be extended by considering renaming and reordering of fragment arguments as described above. The extension to simple fragments is considered here, however the same general principles can be used to extend the ideas to branching fragments.

Recall the definition of a *fragment adaptation* given in Chapter 4; consisting of an instantiation, renaming, variable renaming and input/output argument permutation.

$$\text{FragmentAdapt} == \text{Inst} \times \text{Renaming} \times \text{VarRenaming} \times \text{IOPerm}$$

The definition $\text{matches}_{\text{sfrag}}$ is extended to include variable renamings and reorderings; therefore matching can be defined in terms of fragment adaptations.

$$\begin{array}{|l} \hline \text{matches}_{\text{sfrag}}^* : \mathbb{P}(\text{SimpleFragment} \times \text{SFragQuery} \times \text{FragmentAdapt}) \\ \hline \forall p : \text{SimpleFragment}; q : \text{SFragQuery}; a : \text{FragmentAdapt} \bullet \\ \text{matches}_{\text{sfrag}}^*(p, q, a) \Leftrightarrow \text{let } sp1 = f.\text{spec} \wedge sp2 = \pi_2(q) \text{ in} \\ \text{matches}_{\text{vs}}^*(sp1.\text{invars}, sp2.\text{invars}, \pi_2 a, \pi_3 a, \pi_1(\pi_4 a)) \wedge \\ \text{matches}(\text{renameVars}(sp1.\text{precond}, \pi_3 a), sp2.\text{precond}, \pi_1 a) \wedge \\ \text{matches}_{\text{vs}}^*(sp1.\text{outvars}, sp2.\text{outvars}, \pi_2 a, \pi_3 a, \pi_2(\pi_4 a)) \wedge \\ \text{matches}(\text{renameVars}(sp1.\text{postcond}, \pi_3 a), sp2.\text{postcond}, \pi_1 a) \end{array}$$

Example 7.4 Given the fragment query `adjoin` and fragment `frag` in Fig. 7.5, where `frag` contains parameters f and P . These two specifications can be matched by instantiating the parameters P and f in `frag` under i ; renaming the identifiers in `frag` to match those in `adjoin` with the renaming r ; renaming the I/O variables of `frag` under vr ; swapping the input variables of `frag` (represented by π_{in}) and leaving the position of the outputs fixed (represented by π_{out}):

$$\begin{aligned} i &= \{P(a, b) \rightsquigarrow a \notin b, f(a, b) \rightsquigarrow \{a\} \cup b\} \\ r &= \{X \mapsto \text{Element}, Y \mapsto \text{Set}, Z \mapsto \text{Set}, \text{frag} \mapsto \text{adjoin}\} \\ vr &= \{x \mapsto e, y \mapsto s, z \mapsto r\} \\ \pi_{in} &= \{1 \mapsto 2, 2 \mapsto 1\} \\ \pi_{out} &= \{1 \mapsto 1\} \square \end{aligned}$$

Pattern:

Fragment `frag(x:X,y:Y)` has specification:

precondition $P(x, y)$

output $z:Z$ such that $z = f(x, y)$.

Query:

Fragment `adjoin(s:Set,e:Element)` has specification:

precondition $e \notin s$

output $r:Set$ such that $r = \{e\} \cup s$.

Figure 7.5: Matching fragment specifications

7.4 Discussion

This chapter describes an approach to matching formally specified stand-alone units. The approach is based on matching the corresponding sub-parts of specifications, using a specific algorithm for each kind of specification part, referred to here as matching up to *structural equivalence*. The CARE language is used to illustrate the main concepts.

While the approach is illustrated using units in the CARE language, the approach is general and broadly applicable, provided that the formal specifications of units has a well-defined structure. For example the approach clearly applies to any formal languages where units have a structured *functional specification*. By this we mean components whose behaviour is described by a relationship(s) on the input and output of the component. This is often done in the form of a pre- and post-condition, the former giving a constraint on the values of the input, the latter giving a relationship between the input and output as well as giving a constraint on the outputs.

As an example, the KIDS language has functions described via pre- and post-conditions. For library functions both conditions may contain formal parameters. Matching of functions in KIDS could be done in a similar way to matching of fragments in CARE. That is, firstly attempting to match the inputs and outputs; secondly matching the pre and post-conditions after suitable variable renamings have been performed on the pattern.

The algorithms described in this chapter can inherit either of the matching equivalences described in the previous chapter; i.e. alpha-equivalence or AC-equivalence. Indeed the prototype algorithms for matching units allow the user to select one or the other. Type constrained matching is easily incorporated into unit matching; indeed it is often more useful at the unit level, since the variables which appear in expressions should all be bound by the I/O arguments of units - hence their types will be known. Interactive matching can be extended to the unit level, using a similar framework to that used for expressions. Finally in some cases it may be desirable to parameterise the query, therefore the prototype tools have incorporated unification into the unit matching algorithms.

Zaremski and Wing [97] describe a number of other equivalences such as *guarded post*, that can be used for matching units, which use the semantics of the unit specifications to develop a more sophisticated matching algorithm. While these equivalences are not as general, they are often very useful.

Appeal to the semantics of the CARE language can be used to relax the notion of equivalence for units. For example two fragment specifications may be equivalent if their pre- and post-conditions are logically equivalent. The equivalence can be further relaxed by defining matches between a pattern and query in which the precondition of the pattern is weaker than that of the query, and the postcondition of the pattern is stronger. Another example of a semantic based equivalence is permuting the branches of a branching fragment pattern before attempting to match the query up to structural equivalence. These last two equivalences, referred to as relaxed matching and branching alternatives are described in Chapter 9.