Part III

Retrieval

Chapter 5

A Structured Approach to Component Retrieval Based on Matching

5.1 Introduction

The component-based development paradigm relies on there being a well populated component library. However as the size and complexity of the library grows, it becomes more difficult for the software developer to find suitable components. Retrievability is a measure of how easy it is for the user to find a component meeting their requirements.

The retrievability of components can be improved in a number of ways. Firstly in the design of components and their interfaces: by making the purpose of a component, and its use clear to the developer. Secondly, in the design of the overall library, in particular by paying attention to how components are structured and classified. Finally, by building *retrieval tools* which help the user in finding and adapting suitable components. The focus of this and following chapters shall be on building retrieval tools; the other considerations are outside of the scope of this thesis.

One class of retrieval tools are the query driven retrieval tools, where the user

enters specific requirements (the *search query*) and the tool searches for library components (*patterns*) which match the requirements in some way. There are a variety of query driven retrieval tools including classification-based, information retrieval, signature matching and specification matching. The first two classes of tools are generally applicable to components with informal interfaces, while the other two classes of tools apply to components with formal interfaces.

In comparison to informal languages, formal languages have a number of advantages in describing components. Firstly formal languages are more precise and are less ambiguous. Also formal languages can be used as the basis for building knowledge-based tools - in this case "smarter" retrieval tools. For these reasons this thesis shall focus on retrieval tools based on formal languages.

The solution to retrieval proposed in this chapter unifies adaptation and retrieval. The advantage of this is that retrieval can be used to find not only library components which match the search query in a strict sense, but also those components which "almost" match the query; i.e. matching the query modulo some adaptation. This thesis proposes describing successful matches in terms of an adaptation of the pattern. The advantage of this is that the user doesn't have to provide the input for performing adaptations. Mechanising the adaptation process lessens the chances of errors caused by the user giving an incorrect adaptation.

Section 5.2 contains a general proposal for building retrieval tools. Section 5.3 looks at a variety of formal languages to illustrate how this general proposal for retrieval might work in practice. Finally Section 5.4 presents an overview of how this approach to retrieval will be developed for CARE in the following chapters.

5.2 A general structured approach to retrieval

This section contains a description of a general framework for supporting component retrieval. The framework is a generalisation of other approaches such as signature and specification matching based retrieval, where formal specifications are used as the basis for searching a library of components. The approach described here is

more general than these approaches because it allows for integration with a more general adaptation framework (as opposed to just parameter instantiation), and is applicable to a broader class of components at the unit level (beyond just function-like objects).

There are three main stages in developing such a framework: defining search queries and patterns; developing matching algorithms; and building the overall retrieval tool. The three stages are described separately below.

5.2.1 Defining queries and patterns

The first step in developing a retrieval tool is to define the concepts of *patterns* and *queries* within the context of the component language.

A library is modelled as a set of reusable components.

$[Component]$
$Library == \mathbb{F}\ Component$

It is assumed that each component has an interface which abstractly describes the functionality and behaviour of the component. In the context of retrieval, the individual components in the library will be referred to as *patterns*, reflecting the fact that they can be applied to a variety of problems. To search the library, the user specifies some requirements that prospective patterns must satisfy. This information is encapsulated in the search *query*:

$[Query]$

Remark To maintain the generality of the approach, search queries are modelled separately to the patterns that they are being matched against. In some cases the search query will have the same form as the interfaces of library patterns.

5.2.2 Developing matching algorithms

The next step in developing a retrieval tool is to define the concept of *matches* between patterns and queries. Then algorithms are developed which return a set of matches between patterns and queries based on these definitions.

The solution proposed here for matching patterns and queries is a generalisation of *signature matching* and *specification matching* (see Section 1.5.1 for more details). In particular the approach proposed here goes beyond the scope of just function-like units and modules, and instead considers components more generally. Also the adaptation framework considered here is more general, and considers other techniques beyond just formal parameter instantiation.

A query is said to *match* a library component (pattern) if there exists an adaptation of the pattern which is *equivalent* to the query in some sense - to be made more precise below. As a convention in this thesis, the pattern shall always appear first in the argument list of *matches*, followed by the query, then an adaptation which "achieves" the match.

$$matches : \mathbb{P}(Component \times Query \times Adaptation)$$

The *matches* relation can in general describe an infinite number of tuples: i.e., for any pattern and query there may be an infinite number of matches between the two. Where possible it is useful to define the notion of a *minimal adaptation* in such a way that for any pattern and query there would be a finite set of minimal adaptations, and any other adaptations which describe matches are equivalent to one of the minimal adaptations. Having defined the notion of a minimal adaptation, the next step would be to develop an algorithm to return the (finite) set of adaptations between a given pattern and query.

$$match : Component \times Query \rightarrow \mathbb{F}\, Adaptation$$

If the set of minimal adaptations can be show to be finite the match algorithm is complete (in the sense that it returns representatives of all possible adaptations). Many of the *match* algorithms given below will be complete.

Like the adaptation framework, the approach to matching proposed here is developed by partitioning the languages used to describe component interfaces and queries into three separate levels: expressions, units and modules. Matching can then be defined separately for each level of constructs. From these definitions, algorithms which return a set of matches can be developed. Like adaptation, algorithms for matching constructs in the upper two levels can inherit the algorithms of

the lower level constructs. However the approach gives a clean separation between levels, meaning that alternative algorithms could be substituted without a loss of generality.

5.2.3 Expression matching

To define matching at the expression level let us assume that an abstract syntax for representing patterns and queries has been defined, together with appropriate techniques for adapting expressions. It is also assumed that some notion of equivalence between expressions has been defined. Here a more general case, over and above CARE is considered.

$$[Expr, ExprQuery, ExprAdapt]$$

These constructs have been defined for CARE in Chapters 2 and 4. The *matches* relation will be defined at the expression level. A query matches a pattern if there is an adaptation of the pattern which is *equivalent* to the query. The signature for this relation is as follows:

$$matches : \mathbb{P}(Expr \times ExprQuery \times ExprAdapt)$$

At the expression level, *equivalence* could be as strict as *equality*, or it could be relaxed to other equivalences such as *alpha-equivalence* (see page 73), *AC-equivalence* (see page 142) or *logical equivalence*. In general, a stricter equivalence relation results in less matches between patterns and queries. However less strict equivalences will generally require more sophisticated algorithms, which may include support for formal reasoning.

Having defined the notion of matching for expressions, algorithms which return a set of matches between a pattern and query will be developed.

$$match : Expr \times ExprQuery \rightarrow \mathbb{F}\, ExprAdapt$$

In the case where the patterns are parameterised, support for higher-order logics within the matching algorithm is required.

5.2.4 Unit matching

To define matching at the unit level, let us assume that an abstract syntax for representing patterns and unit queries has been defined, together with appropriate techniques for adapting units. Also assume that some notion of equivalence between units has been defined.

$[Unit, UnitQuery, UnitAdapt]$

A query is said to match a pattern at the unit level if there is an adaptation of the pattern which is equivalent to the query.

$$matches : \mathbb{P}(Unit \times UnitQuery \times UnitAdapt)$$

One form of equivalence at the unit level is *structural equivalence* where the (interface of the) unit pattern after adaptation has the same structure as the unit query, and the corresponding expressions in each are equivalent. The semantics of the language can be used to determine what kind of equivalence is appropriate at the expression level. For example, for function-like units, the semantics of the language may determine that two functions which differ only in their preconditions are equivalent provided their preconditions are logically equivalent. In this case logical equivalence could be used for matching of the preconditions.

Since structural equivalence is very general it is applicable to most formal languages. Other kinds of equivalence, which tend to be less strict can be defined at the unit level. However most of these kinds of equivalences are language-specific, and thus can't be modelled in a general sense. Examples of other forms of unit equivalence for functions are defined by Zaremski [95] (see Section 1.5.1 for more details).

Having defined the *matches* relation for units, a function *match* which returns a set of matches between a unit query and pattern will be defined. The implementation of *match* depends on the equivalence used at the unit level, as well as the particular adaptation framework.

$$match : Unit \times UnitQuery \rightarrow \mathbb{F}\ UnitAdapt$$

5.2.5 Module matching

To define matching at the module level, let us assume that an abstract syntax for representing module patterns and module queries has been defined, together with appropriate techniques for adapting modules. Also assume that the notion of equivalence between modules has been defined.

$$[Module, ModQuery, ModAdapt]$$

A module pattern is said to match a module query if there is some adaptation of the pattern which is equivalent to the query.

$$matches : \mathbb{P}(Module \times ModQuery \times ModAdapt)$$

Matching of modules is done by matching the individual units in the pattern and query. Matching two sets of units can be done in a number of ways; therefore a number of equivalences can be defined. Three ways of matching sets of units are explored in this thesis. Firstly *ALL-match* where all of the unit queries must match a unit in the pattern. Secondly *SOME-match* where at least one of the unit queries must match a unit in the pattern. Thirdly *ONE-match* where exactly one of the unit queries is matched against a unit in the pattern. The motivation for each of these equivalences will be explained later.

Matching algorithms will be developed for each of these equivalences. Each of these algorithms have the following signature.

$$match : Module \times ModQuery \rightarrow \mathbb{F}\, ModAdapt$$

5.2.6 Building a retrieval tool

Having defined the notion of matches between queries and components, and developed algorithms which return a set of matches, the final step is to build a retrieval tool. A top level specification for such a retrieval tool, defined in terms of the *matches* relation is given as follows:

$$search : Query \times Library \rightarrow \mathbb{F}(Component \times Adaptation)$$

$$\forall\, q : Query;\ l : Library;\ c : Component;\ a : Adaptation \bullet$$
$$(c, a) \in search(q, l) \Rightarrow c \in l \land matches(q, c, a)$$

The search tool could be implemented by calling the appropriate match algorithm, which returns a set of matches between the query for each pattern in the library. These sets of matches are merged to form the final result.

A retrieval tool can be made more efficient and effective by enhancing it with techniques which help "narrow the search space". Narrowing the search space means that the (full) matching algorithms only need to be applied to a subset of the library. As a result, the number of patterns that the matching algorithm is applied to is reduced, leading to a more efficient search; and/or the number of matches returned by the search is reduced, leading to a more effective search (in the sense that the user has an easier choice, provided suitable matches are returned).

Techniques for narrowing the search space should in effect be simpler and more efficient than the overall matching algorithm. Such techniques can be integrated into the retrieval tool, either by enhancing the appropriate matching algorithm, or by building separate pre- or post-processing tools. This will be explained in Chapter 9.

Note that while the idea of narrowing the search space is general, the techniques used may be specific to the formal language used; i.e. those techniques based on the semantics of the formal language.

5.3 Retrieval examples

This section illustrates how the general retrieval framework proposed in the previous section might work in practice by looking at a variety of formal languages.

5.3.1 Theories

The libraries of theorem provers such as HOL [1] and Isabelle [2] contain a number of theories, each consisting of operator declarations and assertions of various kinds (including theorems, axioms, rules etc.). A tool for searching for suitable theories in the library is a useful adjunct to a theorem prover, in that it can provide user assistance, or indeed automate parts of the proof.

For searching theories, a number of different kinds of queries are useful. One

possibility is to use signatures of operators (such as functions, relations etc.), which may appear in a theory. Such a query would *match* a theory if there was an operator declared (or used) in the theory whose signature is equivalent to the query (up to instantiation of type parameters). Depending on the semantic foundations of the type system, equivalence may be as simple as equality, or it may be more sophisticated.

Another possible form of query is to use an expression (which may for example correspond to a line in a proof). This kind of query *matches* a theory if the query is equivalent to an assertion in the theory (up to instantiation of formal parameters). The context of the query will determine which equivalence relation is appropriate. In some cases the equivalence may be logical equivalence; in other cases it might be relaxed to showing that the query is implied by the pattern.

Finally, since theories are module-like, the query may consist of a number of individual unit queries. This kind of query matches a theory if each unit in the query matches a unit in the theory.

Example 5.1 Suppose a search query consists of three rules for empty sets, as given in Fig. 5.1. Then searching the Isabelle theory library with this query returns a match against the three rules `subset_refl`, `subset_antisym` and `subset_trans` in the `Set` theory, from the HOL object logic (defined on pages 245-251 of [73]. The relevant parts pertaining to subset and equality relations is repeated in Fig. 5.2).

The three rules in the query match the rules in the library pattern, with instantiation of parameters $A \rightsquigarrow \{\}$, $B \rightsquigarrow X$ and $C \rightsquigarrow Y$.□

```
{} <= X
[| {} <= X; X <= {} |] ==> {} = X
[| {} <= X; X <= Y |] ==> {} <= Y
```

Figure 5.1: A query for searching Isabelle theories

```
subsetI         (!!x.x:A ==> x:B) ==> A <= B
subsetD         [| A <= B; c:A |]  ==> c:B
subsetCE        [| A <= B; ~(c:A) ==> P; c:B ==> P |] ==> P

subset_refl     A <= A
subset_antisym  [| A <= B; B <= A |] ==> A = B
subset_trans    [| A <= B; B <= C |] ==> A <= C

set_ext         [| !!x. (x:A) = (x:B) |] ==> A = B
equalityD1      A = B ==> A <= B
equalityD2      A = B ==> B <= A
equalityE       [| A = B; [| A <= B; B <= A |] ==> P |] ==> P

equalityCE      [| A = B; [| c:A; c:B |] ==> P;
                          [| ~ c:A; ~ c:B |] ==> P
                |] ==> P
```

Figure 5.2: The subset and equality relations in Isabelle

5.3.2 Z Schemas

Since (standard) Z doesn't support modules, libraries would typically consist of stand-alone constructs such as axiom definitions and schemas.

The simplest form of query for searching a Z library would be the declaration part of an axiom definition or schema. For axiom definitions this corresponds to the signature(s) of the constructs being defined. For schemas, the query would consist of a declaration of the state variables, and any inputs/outputs to the schema. A query matches a library pattern if the declaration part of the pattern is equivalent to the query (up to some adaptation of the pattern). Two axiom definitions are equivalent if their signature parts are *type equivalent.* Two schemas are equivalent if they have the same state variables and the signatures of corresponding state variables are equivalent.

Example 5.2 An example search query for Z schemas is given in Fig. 5.3. This search query matches the schema *Inventory* defined in Fig. 5.4. The adaptation required to provide such a match involves: instantiating the formal parameter $STOCK$ to $FOODSTUFF$; renaming the variables $price \leadsto cost$ and $available \leadsto instock$, and

the schema name *Inventory* to *Merchandise*; as well as swapping the first and second lines in the schema. □

$$
\begin{array}{|l}
\textit{Merchandise} \\
\hline
cost : FOODSTUFF \rightarrow\!\!\!\!\!+ COST \\
instock : \mathbb{P}\, FOODSTUFF \\
quantity : FOODSTUFF \rightarrow\!\!\!\!\!+ \mathbb{N}_1 \\
\hline
\end{array}
$$

Figure 5.3: A search query in Z

$$
\begin{array}{|l}
\textit{Inventory}\,[STOCK] \\
\hline
available : \mathbb{P}\, STOCK \\
price : STOCK \rightarrow\!\!\!\!\!+ COST \\
quantity : STOCK \rightarrow\!\!\!\!\!+ \mathbb{N}_1 \\
\hline
\mathrm{dom}\, price = available \\
\mathrm{dom}\, quantity = available \\
\hline
\end{array}
$$

Figure 5.4: A generic schema for representing a store inventory

A more sophisticated search query could also include a number of predicates involving the schema arguments. Matching such queries against patterns from the library would involve matching the declaration part (as described above), as well as matching the predicate parts. A strict predicate matching approach would require that each predicate in the query has a logically equivalent counterpart in the pattern. A relaxed approach might however allow the query predicates to be more general (weaker) than their pattern counterparts or visa-versa. Alternatively it may be sufficient for some (not all) of the query predicates to be matched.

5.3.3 Object-Z

Libraries for supporting Object-Z might contain *classes*, which consist of one or more units such as schemas, axiom definitions, initialisations, etc. Queries in Object-Z could therefore consist of the declaration parts of units, similar to the queries

which might be used in Z; however in this case the query would consist of a set of such constructs. A query matches a pattern, if each of the unit queries match the declaration part of one of the units in the pattern.

Example 5.3 Consider the search query given in Fig. 5.5, consisting of two units - a state schema, and an operation schema. This can be matched against the *Stack* class given in Fig. 5.6, with the query unit *AddElem* matching *Push*.□

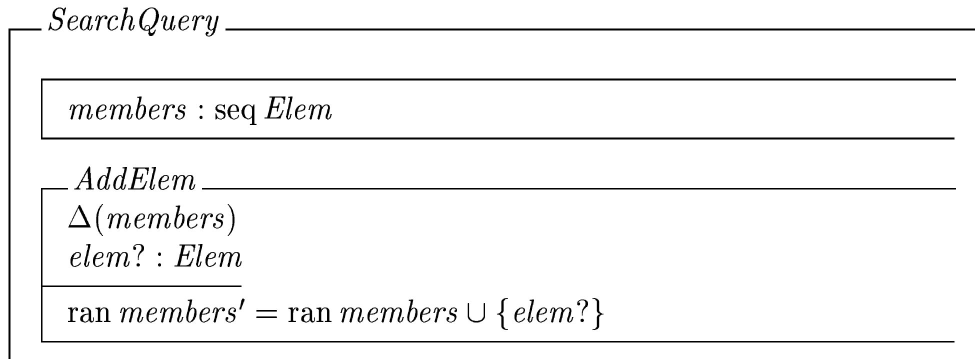

Figure 5.5: A search query for Object-Z

5.4 Development of a retrieval tool in CARE

In the remainder of Part III of the thesis, a retrieval tool is developed for CARE, based on the framework proposed in this chapter. Formal treatments of expression matching, unit matching and module matching are covered in separate chapters, as is the description of a generic retrieval tool.

Chapter 6 contains a number of algorithms for matching expressions in CARE. Firstly algorithms for matching up to alpha-equivalence are developed for different kinds of expressions. Next the equivalence is relaxed to AC-equivalence, and a description of algorithms for matching up to AC-equivalence is given. Two techniques which exploit the semantics of the language to narrow the search space are explored - type-constrained matching and interactive matching. Finally a method for unifying two parameterised expressions is described.

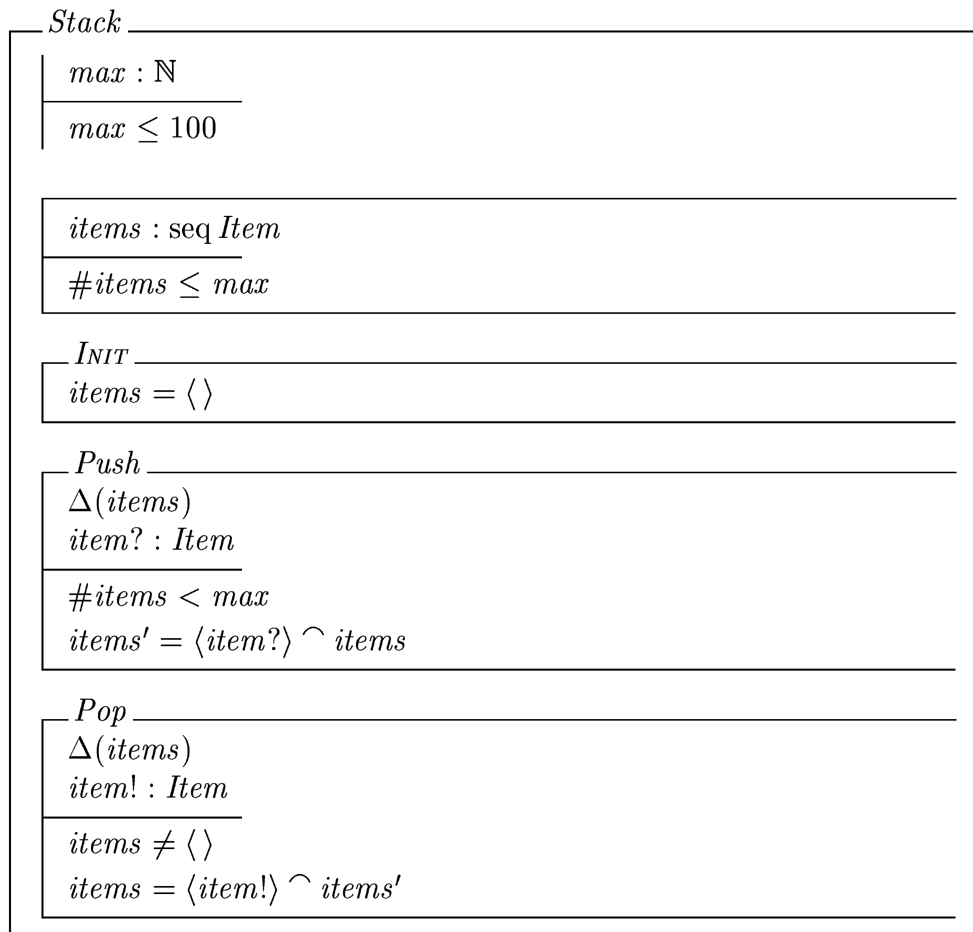

Figure 5.6: A class for stacks in Object-Z

Chapter 7 contains a number of algorithms for matching units in CARE. These algorithms are based on *structural equivalence* of the pattern and the query. The algorithms are implemented by using the expression matching algorithms from Chapter 6; however it is noted that other expression matching algorithms could easily be substituted without loss of generality.

In Chapter 8 a number of algorithms for matching modules are given. These algorithms are general, and should be applicable to most module-like components, including templates in the CARE language. The search query for module matching is a set of unit specifications and module matching is based on matching individual query units against the units of a module. A number of different strategies are given: matching all query units; matching some query units; and matching exactly

one query unit.

At the end of Chapter 8 a general *search engine* is described, which uses the different module matching strategies, and in turn the different expression matching and unit matching algorithms. The search engine is configurable to the user's needs, and can be adapted to build a variety of application-specific retrieval tools.

Chapter 9 contains an illustration of how this search engine can be configured to build a library retrieval tool for CARE.