SOFTWARE VERIFICATION RESEARCH CENTRE

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 02-34

Specification matching of state-based modular components

David Hemer

September 2002

Phone: +61 7 3365 1003
Fax: +61 7 3365 1533
http://svrc.it.uq.edu.au

Note: Most SVRC technical reports are available via anonymous ftp, from `svrc.it.uq.edu.au` in the directory `/pub/techreports`. Abstracts and compressed postscript files are available via `http://svrc.it.uq.edu.au`

Specification matching of state-based modular components

David Hemer

Abstract

Retrieval of software components from a library relies on techniques for matching user requirements against library component interfaces. In this paper we introduce a number of techniques for matching formally specified, state-based modules. These techniques will form the basis for retrieval tool support. The techniques described in this paper build on existing *specification matching* methods, based on individual functions, specified using pre- and post-conditions. We begin by defining a basic module matching technique, based on matching the individual units within a module. We consider variations of this technique that take into account two important features of modules: the visibility of module entities; and the use of state invariants. An advanced technique, based on data refinement and the use of coupling invariants, is also described.

Keywords: Component-based development, Retrieval, Specification matching

1 Introduction

As an answer to the "software crisis" the idea of component-based development was proposed [McIlroy, 1969]. The idea is analogous to building a complex electronic component from a number of smaller, simpler, well-known components in an electronic engineering context. The engineer browses a catalogue of component descriptions for suitable components which can be pieced together in some manner to build a complex component. To build this component it may be necessary to modify the catalogue components in some way.

Given a high-level formal specification of program requirements, compilable code is generated by matching specification statements against a library of dependable program components [Fidge, 2001]. The components, referred to as *templates*, are represented as refinements between a specification and an implementation, represented using Dijkstra's guarded command language [Dijkstra, 1976].

Templates are defined as modular, state-based components, containing a number of operations consisting of a specification and an implementation [Hemer, 2002]. Templates may include higher-order parameters that can be instantiated to solve a variety of problems.

One of the main challenges, besides populating the library, is developing tools for retrieving suitable library components given a specification of the requirements. In this paper we propose using *specification matching* [Zaremski and Wing, 1996], a technique for comparing formal specifications, as a basis for developing retrieval support. A common specification language is used to specify the program requirements (the *query*), and library components.

The specification (interface) part of our template language is very similar to the Sum specification language [Traynor et al., 1995] (a modular extension of the Z specification language [Spivey, 1989]), with the exception of a couple of advanced features. Given that Sum should be easy to understand for those readers familiar with the Z specification language, we will use it to investigate specification matching techniques for state-based modules. It is straightforward to transfer these matching techniques over to our template language.

In Section 2 we give an overview of specification matching, describing three matching techniques relevant to this paper. Section 3 describes the Sum specification language, illustrating the language with some simple examples. Section 4 defines a technique

for matching state-based modules. This section also defines two extensions to the basic technique; the matching methods are illustrated with an example. Section 5 defines an advanced module matching algorithm, based on the ideas of data refinement and using coupling invariants.

2 Specification matching

Retrieving components that satisfy the requirements of the engineer from a component library is one of the challenges of component-based development. In general retrieval relies on algorithms for matching library component specifications against user requirements. There are many approaches that work with informal interfaces, such as keyword based retrieval, and classification based retrieval. However it is difficult to capture user requirements precisely and in an unambiguous manner using these approaches.

A promising approach that alleviates some of these problems is using formal languages to specify the component interfaces and user requirements. Retrieval is based on matching these formal specifications. We will refer to such approaches broadly as *specification matching*, although typically specification matching refers to a particular class of specifications (functions specified using pre- and post-conditions).

A number of different approaches to specification matching have been proposed [Zaremski and Wing, 1996]. Each of these approaches matches a query Q against a library component specification S. The query and library component specification use the same specification language, specified using a pre-condition and post-condition. For the query Q the pre-condition is denoted as Q_{pre}, while the post-condition is denoted as Q_{post}. Similar naming conventions are used to refer to the pre- and post-condition of the library component.

We will briefly describe three specification matching techniques relevant to this paper: exact matching; plug-in match; and guarded plug-in match. For more details on these and other techniques the reader is referred elsewhere [Zaremski and Wing, 1996].

A query and library component are *exact matches* iff their preconditions are logically equivalent, and their postconditions are logically equivalent.

Definition 2.1 (Exact Pre/Post match) *A query Q and library component S are said to be an* exact match*, iff*

$$(Q_{pre} \Leftrightarrow S_{pre}) \wedge (Q_{post} \Leftrightarrow S_{post})$$

Plug-in match succeeds when the precondition of the library component is weaker than that of the query, and the postcondition of the library component is stronger than that of the query.

Definition 2.2 (Plug-in match) *A query Q and library component S are said to be a* plug-in match *iff*

$$(Q_{pre} \Rightarrow S_{pre}) \wedge (S_{post} \Rightarrow Q_{post})$$

Guarded plug-in match is based on plug-in match, but adds the precondition of the library component to the post-condition relation.

Definition 2.3 (Guarded plug-in match) *A query Q and library component S are said to be a* guarded plug-in match *iff*

$$(Q_{pre} \Rightarrow S_{pre}) \wedge ((S_{pre} \wedge S_{post}) \Rightarrow Q_{post})$$

The above techniques are used to match individual units. They can be extended to the module level (containing a collection of units) by defining a query to be a set of user requirements [Zaremski and Wing, 1996]. Modules are specified by specifying each of the individual units in the module. A query matches a module if *all* query requirements are matched against a module unit specification. There are also additional algorithms for matching *some* of the query requirements, and matching *exactly one* of the query requirements [Hemer and Lindsay, 2001].

3 Sum modules

For the purpose of illustrating the module matching techniques described later in this paper we will adopt

the Sum specification language [Traynor et al., 1995]. We choose Sum because of its similarity to the Z specification language [Spivey, 1989] which is fairly widely known and used. We also choose the Sum language because it captures most of the specification features of our template language [Hemer, 2002].

The Sum specification language is similar to the Z language [Spivey, 1989] with several notable differences. Most importantly, from the point of view of this paper, is that Sum allows modules to be specified, and it allows explicit preconditions to be included in operations. Furthermore it syntactically distinguishes between state schemas, initialisation schemas and operation schemas.

Consider the Stack module specification given in Fig. 1. The module is parameterised over the set of values E that the stack can hold. The state of the stack, represented by the *State* schema, consists of the state variable *elements* of type sequence of elements, representing the contents of the stack, as well as the variable *count* representing the number of elements in the stack. An invariant is given for the state indicating that the stack count corresponds to the number of elements in the stack.

The stack is initialised, within the *Init* schema, by setting *elements* to be the empty sequence and setting the *count* to zero. Operations are provided for pushing an element onto the top of the stack, popping an element off of the stack and retrieving an element from the top of the stack. The operation *Push* has an argument *element*?, where the "?" indicates that the argument is an input to the operation. The state is updated by attaching *element*? to the front of the stack and incrementing the count by one. The operation *Pop* has an argument *element*!, where the "!" indicates that the argument is an output of the operation. A precondition is given for the *Pop* operation stating that the stack cannot be empty. The state is updated by removing *element*! from the front of the stack and decrementing the count by one. The operation *Top* has a precondition stating that the stack must be non-empty, and returns the element at the front of the stack. The state remains unchanged by this operation.

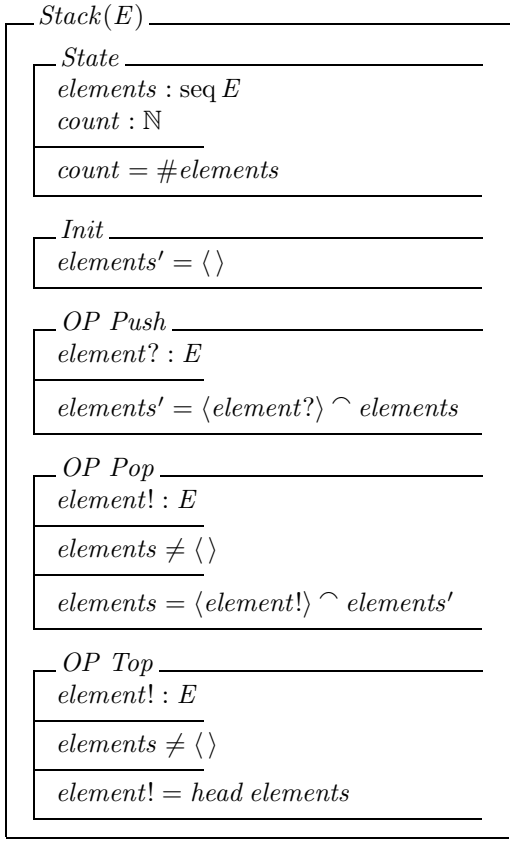

Figure 1: Specification module for stacks

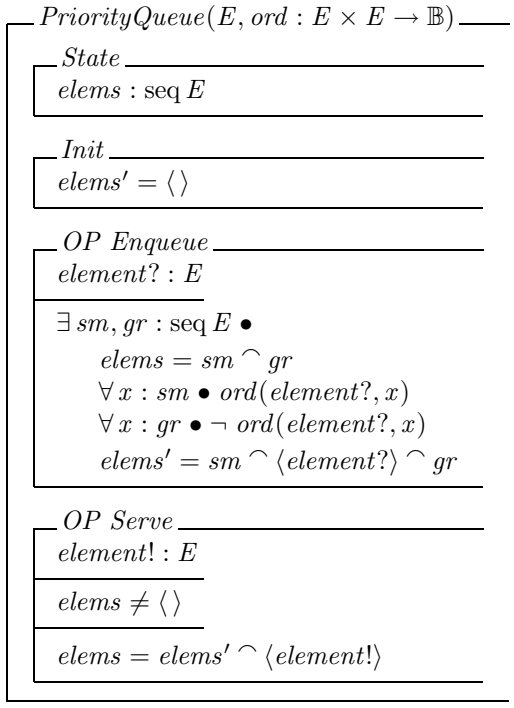

Figure 2: Library specification module for sorted queues

3.1 Parameterisation

Modules can be parameterised over a restricted set of entities. These entities include given sets, operations and functions (relations are modelled as functions that map to boolean values). For example the *PriorityQueue* module specification shown in Fig. 2 is parameterised over the set of values E that the queue may hold, as well as a (partial) ordering, *ord*, over this set. To use this module, both of these parameters must be instantiated. For example we might instantiate E to the set of natural numbers $\mathbb{N}$ and instantiate *ord* to the usual ordering $<$ over natural numbers. Alternatively we could instantiate E to the set of words, and instantiate *ord* to the lexicographical ordering on words.

3.2 Importing modules

To reference a module in a program it must first be *imported*. Import brings into scope the entities declared in the referenced module. An import command specifies the module to be imported, and must also provide an instantiation for all of the module parameters (Sum does not allow partial instantiation of module parameters). The following command imports the *Stack* module, instantiating the set E to the set of natural numbers $\mathbb{N}$.

UseStack
import *Stack*($\mathbb{N}$)

Import commands can also rename any of the entities that appear in the module, including state variables, operations, functions and relations. The following import command has the same effect as the previous command, except it also renames the state variable *elements* to *stacknats*, the operation *Push* to *PushNat*, and the operation *Pop* to *PopNat*.

UseStack
import *Stack*[*stacknats*/*elements*, *PushNat*/*Push*, *PopNat*/*Pop*]($\mathbb{N}$)

3.3 Visible entities

In general module entities contained in imported modules are referenced by means of qualified names. However Sum provides a mechanism, using the *visible* clause, where module entities can be accessed directly without the need for qualified names. The following command imports the *Stack* module and makes all entities provided by *Stack* directly accessible without the need for qualified names.

UseStack
import *Stack*($\mathbb{N}$);
visible *Stack*;

Sum also allows selective entities within a module to be made visible, thus providing a mechanism for narrowing the scope of a module specification. For example the following command imports the *Stack*

modules as before, but this time only the *Pop* operation is directly accessible; the other entities can only be referenced using qualified names.

UseStack

import $Stack(\mathbb{N})$;
visible $Stack[Pop]$;

4 Module matching

4.1 Unit matching

To match modules, we require methods for matching the individual units that appear in modules. In this paper we will restrict our attention to three kinds of units: state schemas; initialisation schemas; and operation schemas. The notion of units differs slightly from entities as used in Sum. Entities can have a smaller level of granularity. For example individual state variables are regarded as entities, however only the complete collection of state variables is considered a unit.

We will define requirements for matching each of these kinds of units separately; as would be expected we do not attempt to match units of a different kind. Following previous naming conventions [Zaremski and Wing, 1996], Q will refer to the query and S will refer to the library component specification.

We assume that for all variables, the function *name* returns the name of a variable, and the function *type* returns the type of a variable.

State schemas The state schemas Q and S match iff any state representable by Q can also be represented by S. More precisely, suppose the state schema Q defines state variables Q_{var} and invariant Q_{inv}, and S defines state variables S_{var} and invariant S_{inv}. Then the state schemas match with respect to a renaming, σ, of state variables in S and an instantiation, π, of parameters in S iff there is an injective mapping $f : Q_{var} \rightarrowtail S_{var}$ such that for all variables v in the domain of f

1. The names of the correspond state variables are the same under σ, i.e.,

$$name(v) = name(f(v)[\sigma])$$

2. The type for each state variable in Q is a subtype of the corresponding state variable in S, i.e.,

$$type(v) \subseteq type(f(v)(\pi))$$

and for invariants S_{inv} and Q_{inv} of S and Q respectively:

$$\forall\, Q_{var} \bullet Q_{inv} \Rightarrow \exists\, S_{var}(\pi) \bullet S_{inv}[\sigma](\pi).$$

For example, the query Q and specification S given below match with respect to the renaming $[nats/ints, lcchars/chars]$.

QState

$nats : \{n : \mathbb{N} \mid n \geq 5\}$
$lcchars : \{c : CHAR \mid islowercase(c)\}$

SState

$chars : CHAR$
$ints : \mathbb{Z}$

To achieve the match *nats* is mapped to *ints*, observing that $\{n : \mathbb{N} \mid n \geq 5\} \subseteq \mathbb{Z}$. Similarly *lcchars* is mapped to *chars*, observing that the set of lower-case characters is a subset of all characters. In this case the invariants for the query and library specification are both *true*, so the matching condition is trivial.

Initialisation schemas The initialisation schemas Q and S match iff there is a renaming, σ, of the state variables in S and an instantiation, π, of the parameters in S such that the initialisation condition, S_{init} of S, adapted with respect to σ and π, implies the initialisation condition, Q_{init}, of Q, i.e.,

$$S_{init}[\sigma](\pi) \Rightarrow Q_{init}$$

Operation schemas Given the operation Q, with a local variable set Q_{lvar}, pre-condition Q_{pre} and post-condition Q_{post}; and the operation S, with a local variable set S_{lvar}, pre-condition S_{pre} and post-condition S_{post}; then Q is said to match S with respect to a renaming, σ, and instantiation, π, iff

1. the name of S is renamed to the name of Q under σ;
2. there is some renaming ρ of the variables in S_{lvar}, and a mapping $f : Q_{lvar} \rightarrowtail\!\!\!\!\!\rightarrow S_{lvar}$, such that for all variables v in Q_{lvar}
   (a) the names of corresponding local variables are the same under ρ, i.e., $name(v) = name(f(v)[\rho])$
   (b) the type of each variable in Q is a subtype of the corresponding variable from S, with respect to the instantiation π i.e.,
   $$type(v) \subseteq type(f(v)(\pi))$$
3. the pre- and post-conditions Q_{pre} and Q_{post}, match the pre- and post-conditions $S_{pre}[\rho \cup \sigma](\pi)$ and $S_{post}[\rho \cup \sigma](\pi)$, adapted with respect to σ, ρ and π using either exact match, plug-in match or guarded plug-in match, as described in Section 2.

4.2 Basic module matching

Module matching can be defined by matching individual module units using the unit matching techniques described above.

Definition 4.1 (Basic module match) *A query Q matches a library component specification S if there is a renaming σ of entities from S and an instantiation, π, of the parameters of S, such that every unit in Q matches a distinct unit from $S[\sigma](\pi)$ using one of the unit matching strategies described in Section 4.1.*

To illustrate basic module matching, consider the search query, *Query*, shown in Fig. 3, which encapsulates the user's requirements for a data structure

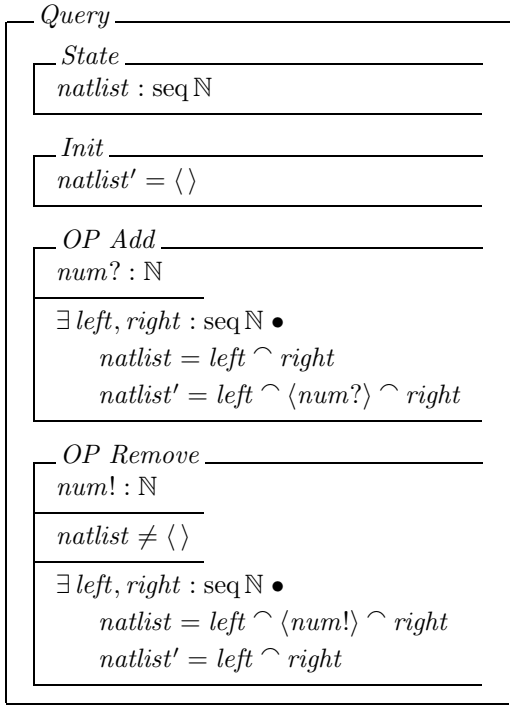

Figure 3: Query specification

representing a sequence of naturals, together with operations for adding a number to the sequence, and an operation for removing one number from a non-empty sequence. This can be matched against the *Stack* module, shown in Fig. 1, using the basic matching strategy. The match is achieved by renaming $[natlist/elements, Add/Push, Remove/Pop]$, and instantiating the parameter E to the set $\mathbb{N}$.

It is simple to show that the state schemas match by observing that the state variable *natlist* from *Query* can be represented by the state variable *elements* from *Stack*; the *Stack* module also contains the additional state variable *count*. The proof obligation for the state invariants is

$$\forall natlist : \mathrm{seq}\,\mathbb{N} \bullet \mathrm{true} \Rightarrow \exists elements : \mathrm{seq}\,\mathbb{N};\ count : \mathbb{N} \bullet \#natlist = count$$

This condition is simple to prove. To show that the initialisation schemas match we need to discharge the following proof obligation:

$$(natlist' = \langle\,\rangle) \Rightarrow natlist' = \langle\,\rangle$$

This is trivial to prove.

The operation *Add* from the *Query* module can be matched against operation *Push* from the *Stack* module by renaming the local variable *element*? to *num*?, and observing that the post-condition of *Add* is stronger than the post-condition of *Push*, i.e.:

$$\begin{array}{l}(natlist' = \langle num? \rangle \frown natlist) \Rightarrow \\ \quad \exists\, left, right : \operatorname{seq} \mathbb{N} \bullet \\ \qquad natlist = left \frown right \wedge \\ \qquad natlist' = left \frown \langle num? \rangle \frown right\end{array}$$

The operation *Remove* can be matched against the operation *Pop* by: renaming the local variable *num*! to *element*!; observing that the pre-condition of *Remove* is equivalent to that of *Pop*, i.e.,

$$natlist \neq \langle\,\rangle \Rightarrow natlist \neq \langle\,\rangle$$

and observing that the post-condition of *Remove* is stronger than that of *Pop*, i.e.,

$$\begin{array}{l}(natlist = \langle num! \rangle \frown natlist') \Rightarrow \\ \quad \exists\, left, right : \operatorname{seq} \mathbb{N} \bullet \\ \qquad natlist = left \frown \langle num! \rangle \frown right \\ \qquad natlist' = left \frown right\end{array}$$

With this match the following program using the *Query* module:

Program

import *Query*;
visible *Query*;
...

can be replaced by a program that uses the stack module, as follows:

Program

import *Stack*[*natlist*/*elements*,
Add/*Push*, *Remove*/*Pop*](ℕ);
visible *Stack*;
...

4.3 Entity subset matching

One problem with the basic module matching method, as the above example illustrates, is that it makes all entities within the library component module visible. In general the library component will contain entities that are surplus to the requirements of the query and thus is sometimes safer to keep these operations protected. In the example above, the library component included the additional state variable *count*, and the additional operation *Top*.

Definition 4.2 (Entity subset module match) *A library component S matches a query Q, with respect to a subset, η, of the entities in S, an entity renaming σ and an instantiation π iff*

1. *If S^{η}_{var} represents the state variables of S contained in η and Q_{var} represents the state variables of Q, then there is a bijection $f : Q_{var} \rightarrowtail\!\!\!\!\!\rightarrow S^{\eta}_{var}$ such that for all variables v in the domain of f*
   (a) *$name(v) = name(f(v)[\sigma])$*
   (b) *$type(v) \subseteq type(f(v)(\pi))$*
   (c) *the invariants match as before*
2. *The initialisation schemas match as before*
3. *Each operation of Q matches an operation of S that is named in η with respect to σ and π, using either exact matching, plug-in matching or guarded plug-in matching.*

Using this matching strategy to match the search query, *Query*, and the stack module, *Stack*, we would get the same renaming and instantiation, but would also get the entity subset $\{elements, Pop, Push\}$ indicating the entities that should be made visible. In this case *Program* would be refined to the following program:

Program

import *Stack*[*natlist*/*elements*,
Add/*Push*, *Remove*/*Pop*](ℕ);
visible *Stack*[*elements*, *Pop*, *Push*];
...

4.4 Using state invariants

Another improvement to module matching is to use state invariants as contextual information when doing matching. The idea is similar to guarded plug-in matching, where the pre-condition of the library component is assumed to show that the post-condition of the query and library components match. In this case the state invariant of the library component is assumed in matching both the pre-conditions and post-conditions.

Guarded plug-in match can be redefined as follows to make use of the state invariant, S_{inv} of the library component S.

Definition 4.3 (Guarded Inv plug-in match) *A query Q and library component S are said to be an* guarded invariant plug-in match *iff*

$$((Q_{pre} \land S_{inv}) \Rightarrow S_{pre}) \land \\ ((S_{pre} \land S_{post} \land S_{inv}) \Rightarrow Q_{post})$$

This operation matching method could be used in addition to the other matching methods defined in Section 2 in the two module matching algorithms described earlier.

To illustrate how such a method would be useful, suppose that the *Pop* operation from the *Stack* module, in Fig. 1, used *count* instead of *elements* to ensure that the stack was non-empty in the pre-condition, i.e.,

OP Pop

$element! : E$

$count > 0$

$elements = \langle element! \rangle \frown elements' \land$
$count' = count - 1$

To match this using plug-in or guarded plug-in match against the operation *Remove* from Fig. 3, we would be required to prove (after renamings), that the pre-condition of the library operation is weaker than that of the query, i.e.,

$$natlist \neq \langle\,\rangle \Rightarrow count \neq 0$$

This is clearly not provable; however using guarded invariant plug-in match we can also assume the invariant, therefore the proof obligation becomes:

$$(natlist \neq \langle\,\rangle \land count = \#natlist) \Rightarrow count > 0$$

which is straightforward to prove (assuming the length function, #, maps a sequence of a natural number, i.e., $\# : (\mathrm{seq}\, E) \rightarrow \mathbb{N}$).

5 Advanced matching

The module matching strategies described to date [Zaremski and Wing, 1996, Hemer and Lindsay, 2001] have been restricted to matching data structures with the same underlying type (or a subtype). In this section we propose an advanced matching technique, where query modules that define an abstract data structure are matched against library components that implement a more concrete data structure, and where the concrete data structure can be used to represent the abstract one. The idea is based on data refinement [Morgan, 1994, Back and von Wright, 1998], where operations on the abstract data structure are implemented by operations on the concrete data structure. A simple example is representing a set as a list, and using the list operations to implement the set operations.

Like data refinement, the module matching method described in this section relies on a coupling invariant that describes the relationship between abstract and concrete data representations. We begin by defining methods for matching individual module units, and then use these methods to define and overall module matching strategy.

5.1 Operation matching

The following definitions are based on previous specification matching methods [Zaremski and Wing, 1996] (repeated in Section 2). In these definitions Q_{in} (S_{in}) refers to the initial (unprimed) state variables of Q (S). Similarly Q_{out} (S_{out}) refers to the final (primed) variables of Q (S). The coupling invariant CI is a relationship between

the state variables (either initial or final) of Q and S.

Definition 5.1 (Exact Pre/Post CI match) *A query Q and library component S are said to be an exact pre/post match, with respect to a coupling invariant CI, iff*

$$((CI(Q_{in}, S_{in}) \land Q_{pre}) \Leftrightarrow S_{pre}) \land \\ ((CI(Q_{in}, S_{in}) \land CI(Q_{out}, S_{out}) \\ \land S_{post}) \Leftrightarrow Q_{post})$$

Definition 5.2 (Plug-in CI match) *A query Q and library component S are said to be a plug-in match, with respect to a coupling invariant CI, iff*

$$((CI(Q_{in}, S_{in}) \land Q_{pre}) \Rightarrow S_{pre}) \land \\ ((CI(Q_{in}, S_{in}) \land CI(Q_{out}, S_{out}) \\ \land S_{post}) \Rightarrow Q_{post})$$

Definition 5.3 (Guarded Plug-in CI match) *A query Q and library component S are said to be a guarded plug-in match, with respect to a coupling invariant CI, iff*

$$((CI(Q_{in}, S_{in}) \land Q_{pre}) \Rightarrow S_{pre}) \land \\ ((CI(Q_{in}, S_{in}) \land CI(Q_{out}, S_{out}) \\ \land S_{pre} \land S_{post}) \Rightarrow Q_{post})$$

5.2 Module matching

Advanced module matching, using coupling invariants, is defined using the operation matching strategies given above.

Definition 5.4 (Advanced module match) *A module query Q is said to match a library module S, iff there is a coupling invariant CI between the state variables of Q and S, and an instantiation, π, of parameters in S such that:*

1. *If the state variables for Q are Q_{var} and the state variables for S are S_{var}. Then each state representable by the query module can be represented by a state from library module, i.e.,*

$$\forall\, Q_{var} \bullet Q_{inv} \Rightarrow \exists\, S_{var}\pi \bullet S_{inv} \\ \land\, CI(Q_{var}, S_{var})$$

2. *The initialisation of S is equivalent with respect to the coupling invariant to the initialisation of Q*

$$S_{init}(\pi) \land CI(Q_{var}, S_{var}) \Rightarrow Q_{init}$$

3. *Each operation of Q matches an operation from S with respect to CI and π using one of the operation matching methods defined in Section 5.1.*

5.3 Example

Suppose we want to represent a "collection" of tasks, represented abstractly as natural numbers, where the number represents their priority (there may be other information but we're not concerned about this here). Furthermore, suppose we require methods for adding a task, and another for accessing the highest priority task. Such a requirement can be encapsulated in a search query, itself a module specification, as shown in Fig. 4.

The *SearchKey* module represents the tasks as a bag (multiset). For example the bag $[\![a, a, b, b, b, c]\!]$ contains two occurrences of the element a, three occurrences of the element b, and one occurrence of c. The state is initialised to the empty bag ($[\![]\!]$). The operations *Add* and *Remove* are specified in terms of the bag union operator $\uplus$; for example $[\![a, a, b]\!] \uplus [\![b, b, c]\!]$ corresponds to the bag $[\![a, a, b, b, b, c]\!]$.

The *SearchKey* module is matched against the *PriorityQueue* module shown in Fig. 2. We begin by instantiating the module parameters E and *ord* to $\mathbb{N}$ and $<$ respectively. Next we need to find a coupling invariant, such that every abstract data structure (from the query) can be represented by a concrete data structure (from the library) with respect to the invariant. Furthermore the initialisation schemas must match with respect to the coupling invariant, and each operation from the query must match an operation from the library module.

We require a coupling invariant between the state variable *tasks* from the (abstract) query module and *elems* from the (concrete) library module. We choose the following coupling invariant:

$$tasks = items(elems)$$

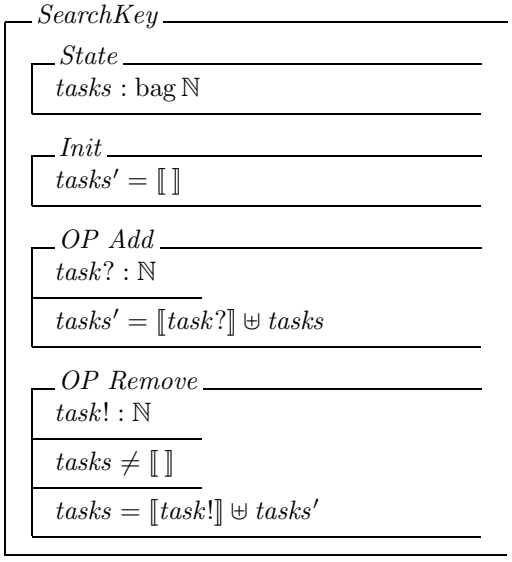

Figure 4: Search query

where *items* is defined as follows:

$$
\begin{array}{|l}
items : \operatorname{seq} E \rightarrow \operatorname{bag} E \\
\hline
items(\langle\,\rangle) = [\![\,]\!] \\
\forall\, e : E \bullet items(\langle e \rangle) = [\![e]\!] \\
\forall\, s, t : \operatorname{seq} E \bullet items(s \frown t) = \\
\qquad items(s) \uplus items(t)
\end{array}
$$

The first matching condition from Definition 5.4 becomes

$$\forall\, x : \operatorname{bag} \mathbb{N} \bullet \exists\, y : \operatorname{seq} \mathbb{N} \bullet items(y) = x$$

The condition for matching the initialisation schemas becomes:

$$
\begin{array}{l}
elems' = \langle\,\rangle \wedge tasks' = items(elems') \Rightarrow \\
\qquad tasks' = [\![\,]\!]
\end{array}
$$

After simplification this becomes

$$items(\langle\,\rangle) = [\![\,]\!]$$

The *Enqueue* and *Add* operations are matched using plug-in CI match. We are required to show the following condition holds:

$$
\begin{array}{l}
\exists\, sm, gr : \operatorname{seq} \mathbb{N} \bullet \\
\qquad elems = sm \frown gr \wedge \\
\qquad \forall\, x : sm \bullet task? < x \wedge \\
\qquad \forall\, x : gr \bullet task? \geq x \wedge \\
\qquad elems' = sm \frown \langle element? \rangle \frown gr \wedge \\
\qquad tasks = items(elems) \wedge \\
\qquad tasks' = items(elems') \Rightarrow \\
\qquad tasks' = [\![task?]\!] \uplus tasks
\end{array}
$$

This can be simplified to:

$$
\begin{array}{l}
items(sm \frown \langle task? \rangle \frown gr) \\
\qquad = [\![task?]\!] \uplus items(sm \frown gr)
\end{array}
$$

This follows from the definition of *items*, and the fact that bag union ($\uplus$) is associative and commutative. The operations *Serve* and *Remove* can be matched using *Guarded Plug-in CI match*. Two conditions result:

$$
\begin{array}{l}
items(elems) = tasks \wedge tasks \neq [\![\,]\!] \\
\qquad \Rightarrow elems \neq \langle\,\rangle
\end{array}
$$

and

$$
\begin{array}{l}
items(elems) = tasks \wedge items(elems') = tasks' \wedge \\
\qquad elems \neq \langle\,\rangle \wedge elems = elems' \frown \langle task! \rangle \\
\qquad\qquad \Rightarrow task = [\![task!]\!] \uplus tasks'
\end{array}
$$

The first of these proof obligations is straightforward. The second can be simplified to the following:

$$items(elems' \frown \langle task! \rangle) = [\![task!]\!] \uplus items(elems')$$

which follows from the definition of *items* and the fact that "$\uplus$" is commutative.

6 Related work

There are a number of existing approaches to specification matching of units. These can be broadly divided into syntactic-based methods and semantic-based methods. The syntactic-based methods [Rollins and Wing, 1991, Hemer and Lindsay, 1997]

use pattern matching or unification to perform structural matching of specifications. Such methods are automatable, however are generally not as powerful as semantic-based matching methods. Semantic-based matching [Zaremski and Wing, 1996, Perry and Popovich, 1993, Jeng and Cheng, 1995] relies on theorem prover support to prove that specifications match. While these methods are more precise than syntactic matching methods, theorem proving is generally very difficult to automate, and therefore becomes a major bottleneck in the retrieval process.

Module specification matching has been previously proposed [Zaremski and Wing, 1996, Hemer and Lindsay, 2001]. However these technique give very little consideration to state-based modules; focusing instead on purely function specification languages.

7 Conclusions

In this paper we describe techniques for matching state-based modules. These techniques will form the basis for developing retrieval tools to support component-based development. The techniques are illustrated using the Sum specification, a modular extensions to the widely used Z specification language. These techniques should be widely applicable to other formal specified modular components.

The challenge remains to develop suitable implementations of these techniques. A key decision needs to made to choose between a syntactic-based matching approach, or a semantic-based matching approach, or perhaps a hybrid approach.

Acknowledgements

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

References

[Back and von Wright, 1998] Back, R. and von Wright, J. (1998). *Refinement Calculus: A Systematic Introduction*. Springer.

[Dijkstra, 1976] Dijkstra, E. (1976). *A Discipline of Programming*. Prentice-Hall. In Series in Automatic Computation.

[Fidge, 2001] Fidge, C. (2001). Compilation of specifications. In *Proceedings.of the Eighth Asia-Pacific Software Engineering Conference (APSEC 2001)*, pages 355–362. IEEE Computer Society Press.

[Hemer, 2002] Hemer, D. (2002). Computer-aided programming using formally specified design templates. In *Proceedings of APSEC'2002*. to appear.

[Hemer and Lindsay, 1997] Hemer, D. and Lindsay, P. (1997). Reuse of verified design templates. In Fitzgerald, J., Jones, C., and Lucas, P., editors, *Formal Methods Europe '97*, number 1313 in Lecture Notes in Computer Science, pages 495–514. Springer.

[Hemer and Lindsay, 2001] Hemer, D. and Lindsay, P. (2001). Specification-based retrieval strategies for module reuse. In Grant, D. and Stirling, L., editors, *Proc. of Australian Software Engineering Conference (ASWEC'2001)*, pages 235–243. IEEE Computer Society.

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

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

[Morgan, 1994] Morgan, C. (1994). *Programming from Specifications*. Prentice Hall, second edition.

[Perry and Popovich, 1993] Perry, D. and Popovich, S. (1993). Inquire: Predicate-based use and reuse. In *Proceedings of the 8th Knowledge-Based Software Engineering Conference*, pages 144–151.

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

[Spivey, 1989] Spivey, J. (1989). *The Z Notation: a Reference Manual*. Prentice-Hall, New York.

[Traynor et al., 1995] Traynor, O., Karlsen, E., Kazmierczak, E., Kearney, P., and Wang, L. (1995). Extending Z with modules. In Kotagiri, R., editor, *Proceedings of the Eighteenth Australasian Computer Science Conference (ACSC'95)*, pages 513–522.

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