

Part II

Adaptation

Chapter 3

Adapting Components for Reuse

3.1 Introduction

In the introduction to this thesis (see Chapter 1) it was noted that for reuse of components to be attractive to the software engineer, the effort in reusing components must be significantly less than the effort required to develop components from scratch. One of the key ways of addressing this principle is helping the user to modify existing components to meet their specific needs.

Components which can be adapted to solve a wide range of problems can lead to a reduction in the size of the library, in comparison to a library of “rigid” components. Perhaps the most obvious advantage of this is the reduction in the amount of storage space that needs to be allocated to the library. Another advantage is the reduction of the search space when attempting to retrieve components from the library, which can lead to more efficient and effective search operations.

The adaptability of a component is a measure of how easy it is to modify a component to suit the user’s requirements. Adaptability is achieved by ensuring that components are designed to have a level of generality, and that the framework for using the reusable components contains methods and tools which help the user to adapt components. For example the designer of a component may choose to achieve a level of generality by parameterising the component. To make effective use of such a component, the framework must include methods and tools for creating instances

of the component, by instantiating the parameters with suitable values.

While it is important to aim for components which can solve a wide range of problems, this goal must be balanced against the simplicity of the component for reuse. A broadly applicable component is not much use if its interface, or the mechanism for adapting the component, are too complicated for the user to comprehend and use effectively. To this end, a compromise between the scope of problems a component can solve, and the component's simplicity must be struck.

As well as containing software, components may have certain *properties* which make them more attractive to the reuser. Examples of component properties include: a component's implementation being correct with respect to its specification; a component having been successfully type-checked; a modular component being self-contained; the implementation part of a component being shown to terminate.

Such properties can add value to a component, but will often take some time for the designer of the component to establish. Indeed establishing such properties will often account for a major part of component design. For these reasons it is clearly desirable that a framework for using reusable components preserve such properties. Most importantly, property preservation must be addressed in designing methods and tools for adapting components.

In Section 3.2 a general, structured approach to component adaptation is formulated. The approach includes a number of adaptation techniques which are applicable across a wide spectrum of formal languages used to express components. In Section 3.3 a number of formal programming and specification languages are used to illustrate how the framework could be applied in the general sense.

3.2 A systematic approach

Component adaptation depends on the language used to represent components and their interfaces; however in this section a general framework for adapting components is presented which exploits a number of commonly occurring features of formal languages.

The first step in developing a framework for adapting a certain kind of component, is to partition the language used to represent component interfaces into three separate tiers. This thesis' approach is to partition the language into *expressions*, *units* and *modules* (as defined earlier in Section 1.7.1). For each of the three separate levels of constructs, methods for adapting the component are identified and operations for applying the methods are developed. These operations are then integrated into a single tool for adapting the components.

The most common method of generalising mathematical expressions is by introducing parameters. Parameterised expressions can be adapted by instantiating any of the parameters which appear in the expression. In developing a reuse framework for components, the kind of parameters which can appear in an expression are identified, and operations for instantiating these parameters are developed.

In general, if a property holds for a parameterised component, then it will also hold for any instances of that component. In some cases however the designer of a component may establish properties that are only valid for some values of the parameters. In this case the designer may include assertions in the component, referred to as *applicability conditions*, which describe the set of values that can be passed to a parameter, such that certain properties are preserved. In this case, in order to establish that properties are preserved, the user of a component would need to show that the applicability conditions are satisfied for the values passed to the parameters.

For units, the kinds of textual parameters which can be renamed without changing the semantics of the unit are identified. This may include function names, data structure names, variable names etc. Next, operations for applying textual parameter renamings to a unit are developed. Also identified are other ways in which the structure of units can be modified without changing the underlying meaning of the unit. Examples of this include reordering the arguments of a function, swapping the order of cases in a case statement, or swapping the order of conditions or schema variable declarations in a Z schema. Operations for applying structural changes to a unit are developed below, which can then be integrated with the renaming

operations.

Modules, which consist of a set of units, can be adapted by restriction to a subset of the original unit set. Such an adaptation technique will be referred to here as *subsetting*. Subsetting is a useful adaptation technique in cases where the user of a module only requires some of the functionality offered by the module.

To incorporate subsetting into the adaptation framework, the notion of a subset of a module (also referred to as a *submodule*) must first be defined. A submodule, should itself be a module, and thus should obey the same syntactic and semantic rules as its parent. Properties which applied to the module should also apply to submodules. For example if a module is self-contained (i.e. any unit used in a module is also defined in the module), then any submodules should also be self-contained. Once the notion of a module has been defined, operations for adapting modules by subsetting can be developed.

The approach proposed here is systematic and applicable to a wide variety of formally specified components. The separate adaptation techniques described are all relatively simple, and are supported by many formal languages. The emphasis is on techniques which can be shown to preserve certain properties.

3.3 Adaptation examples

To illustrate how the adaptation approach could be applied in practice, a number of specification and programming languages are explored. For each of these languages the features which support adaptation are discussed, and examples showing how the components can be adapted are given. The approach is defined in detail on CARE in Chapter 4.

3.3.1 Standard ML

Functional programming languages such as Standard ML [84, 93] support reuse through parametric polymorphism, higher-order types and support for libraries.

Higher-order programming is supported by allowing the arguments of the pro-

gram functions to range over functions. Therefore one means of “adapting” functions is to assign actual values to the arguments. In effect the parameters are being instantiated to create an instance of the function. For example, the function `map`, defined in Fig. 3.1, in which a function `f` is applied to each element of a list, is parameterised over the function `f`.

```
fun map f lis =
  if lis = [] then []
  else f(h(lis))::map (f)(tl(lis));
```

Figure 3.1: The `map` function in Standard ML

To increment by one each element in a list of numbers, the function f could be instantiated as $f(a) \rightsquigarrow a + 1$, where a is a placeholder for the function. Note that the type of the function f would also be (implicitly) instantiated in this case.

Another way of adapting functions in ML is to rename the variables. Since the variables are simply textual parameters, renaming them will not change the meaning of the function, provided they are renamed consistently throughout the scope of their use. Fig. 3.2, shows the result of adapting the `map` function by renaming `f` to `g` and `lis` to `seq`.

```
fun map g seq =
  if seq = [] then []
  else g(h(seq))::map (g)(tl(seq));
```

Figure 3.2: Renaming the variables in the `map` function

Other identifiers, e.g. function names, could also be renamed. For example, suppose that rather than having the functions `h` and `tl` the user has defined similar functions called `head` and `tail` respectively. For the user to make use of the `map` function given in Fig. 3.1, `map` could be adapted by renaming `h` to `head` and `tl` to `tail`. In this case it is important that this kind of renaming is applied consistently throughout the whole scope of the identifiers in question, i.e. wherever the identifier is defined or used.

A further method for adapting functions in ML, is to reorder the function arguments. As a consequence of reordering, wherever the function is called the arguments must also be reordered. For example, if the arguments of the `map` function in

Fig. 3.1 are swapped, then the arguments of the call to `map` must also be swapped. The resulting adapted function is given in Fig. 3.3.

```
fun map lis f =
  if lis = [] then []
  else f(h(lis))::map (tl(lis))(f);
```

Figure 3.3: Reordering the arguments of `map`

3.3.2 Z Schemas

The Z specification language [87] can be partitioned into expressions such as formulae, terms and types, and units including schemas and axiom definitions. Z doesn't support module like structures, although some extensions of Z do (e.g. Sum [50]).

In Z, expressions can contain set parameters which are introduced by defining generic schemas and generic axiomatic definitions. These parameters can be instantiated to create instances of the schema. For example, the generic schema *Inventory* (given in Fig. 3.4) is parameterised over the set *STOCK*, which can then be instantiated to different sets of values, depending upon the target application; e.g. *STOCK* might be instantiated to the set $\{bread, milk, flour, \dots\}$ for a food store.

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

Figure 3.4: A generic schema for representing a store inventory

Schemas can also be adapted by renaming state variables, and provided this is done consistently throughout the scope of these variables, an equivalent specification results. Similarly, the order in which the state variables appear can be rearranged, while preserving the meaning of the schema. For example, the schema

BirthdayBook2 below, is an adaptation of *BirthdayBook1*, where the state variables have been swapped, and the state variable *known* has been renamed to *names*.

<i>BirthdayBook1</i> $known : \mathbb{P} NAME$ $birthday : NAME \rightarrow DATE$
$known = \text{dom } birthday$

<i>BirthdayBook2</i> $birthday : NAME \rightarrow DATE$ $names : \mathbb{P} NAME$
$names = \text{dom } birthday$

3.3.3 C++ Classes

The C++ programming language [90] can be partitioned into three levels. These levels include types and signatures under expressions; operations and data structures under units; and classes under modules.

C++ supports parametric polymorphism by allowing generic classes, referred to as *templates*, to be defined. Fig 3.5 gives an example of the vector class, which is parameterised over the type of the underlying elements. Instances of this class can be created by instantiating the parameter *T* to a concrete predefined or user defined type. For example *T* could be instantiated to `int`, to represent a vector of integers. Alternatively, if the user has defined a class `Sprocket`, then *T* could be instantiated to `Sprocket` to represent a vector of sprockets.

For classes subsetting could be applied by restricting a class to certain operations and data structures. Consider the complex number class given in Fig. 3.6. This class could be adapted by the user to include only those operations that are required. For example the user might only require operations for adding and multiplying two complex numbers, and so would only include the first and fourth operations listed.

```

template<class T> class vector {
    T* v;
    int sz;
public:
    vector(int);
    T& operator[] (int);
    T& elem(int i);
    int size();
    ...
};

```

Figure 3.5: A generic class for vectors in C++

```

class complex {
    double re,im;
public
    complex(double r, double i)
    complex(double r)

    complex operator+(complex,complex);
    complex operator-(complex,complex);    //binary
    complex operator-(complex);            //unary
    complex operator*(complex,complex);
    complex operator/(complex,complex);
};

```

Figure 3.6: Manipulating complex numbers in C++

3.3.4 Theories

Theorem provers, such as HOL [1] and Isabelle [73], support module-like structures, usually referred to as *theories*, in which contain a set of related units such as definitions, rules, constants, type declarations etc. are collected. Theories have an overall theme which connects the individual units in some manner, e.g. a theory for lists or a theory for natural numbers. However, even though the set of constructs in a theory are related, the user may not require all of the functionality offered by a theory for their particular application. For example suppose a general theory for lists is defined which contains constructs for creating an empty list, concatenating two lists, extracting the head and tail of a non-empty list, reversing a list etc., as well as

associated rules and definitions. The user may only require those parts of the theory which relate to concatenating two lists, thus the list theory could be “adapted” to the user’s needs by only including the relevant parts.

Consider the Isabelle theory for natural numbers given in Fig. 3.7. This theory, which inherits the theory of first order logic, contains a type representing natural numbers, as well as constants for representing zero, the successor function, a recursion function and addition. Suppose that the user wants to develop a successor-like theory, and is only interested in the successor function and some associated theory (cf. Fig. 3.8). The user might choose to include the constants 0 and Suc , together with the first three rules.

When including subsets of a theory, it is important to ensure that the subset is self-contained. That is, any construct used in the subset is also defined in the subset. In this case the successor function depends on the rules `Suc_inject` and `Suc_neq_0`, which define properties about the function. In turn the second of these rules, depends on the constant 0 . The exact nature of these dependencies, and how they are defined will depend on the particular application language.

Another point worth noting is that many theorem provers offer support for higher-order logics containing parameters ranging over types, functions and relations. The HOL theorem prover is hardwired to higher order logic, with support for higher-order unification built into the underlying engine. In contrast, Isabelle is a generic theorem prover allowing reasoning to be done in a variety of formal theories. Isabelle does however provide support for higher-order logics. Theories which are based on higher-order logics can be adapted by instantiating the parameters to create instances of the theory.

3.3.5 Object-Z

Object-Z [21] is an object-oriented extension of Z. Like Z, it contains units such as schemas and axiom definitions, which can be parameterised over sets. Modularisation is also supported in the form of *classes*, which encapsulate a number of related state schemas, initialisations, operational schemas and axiom definitions

```

Nat = FOL +
types  nat
arities nat      :: term
consts "0"       :: "nat"                ("0")
      Suc        :: "nat=>nat"
      rec        :: "[nat, 'a, [nat, 'a]=>'a] => 'a"
      "+"        :: "[nat, nat] => nat"    (infix 60)
rules  Suc_inject "Suc(m)=Suc(n) ==> m=n"
      Suc_neq_0  "Suc(m)=0 ==> R"
      induct     "[| P(0); !!x.P(x) ==> P(Suc(x)) |] ==> P(n)"
      rec_0      "rec(0,a,f) = a"
      rec_Suc    "rec(Suc(m),a,f) = f(m,rec(m,a,f))"
      add_def    "m+n == rec(m, n, %x y. Suc(y))"
end

```

Figure 3.7: An Isabelle theory for natural numbers

```

Nat = FOL +
types  nat
arities nat      :: term
consts "0"       :: "nat"                ("0")
      Suc        :: "nat=>nat"
rules  Suc_inject "Suc(m)=Suc(n) ==> m=n"
      Suc_neq_0  "Suc(m)=0 ==> R"
      induct     "[| P(0); !!x.P(x) ==> P(Suc(x)) |] ==> P(n)"
end

```

Figure 3.8: An instance of the natural numbers theory

(amongst other constructs). Therefore, one way of adapting classes in Object-Z is by subsetting.

For example consider the stack class given in Fig. 3.9. It consists of a definition of the constant *max*, a state schema which defines the items in a stack, an initialisation schema, and two operations *Push* and *Pop* which add an element and remove an element respectively from the stack. Suppose however the user only required the *Push* operation. The class could be adapted by including this operation only, however the resulting adapted class must be well-defined. To ensure this is the case, the first three constructs in the class must also be included (i.e. the definition of *max*, the state schema and initialisation schema).

Individual constructs in a class could be adapted in a similar manner to Z constructs, i.e. renaming of identifiers, reordering state variables and instantiating formal parameters in generic schemas/axiom definitions. Note that when renaming identifiers, they must be renamed consistently throughout the scope of their use.

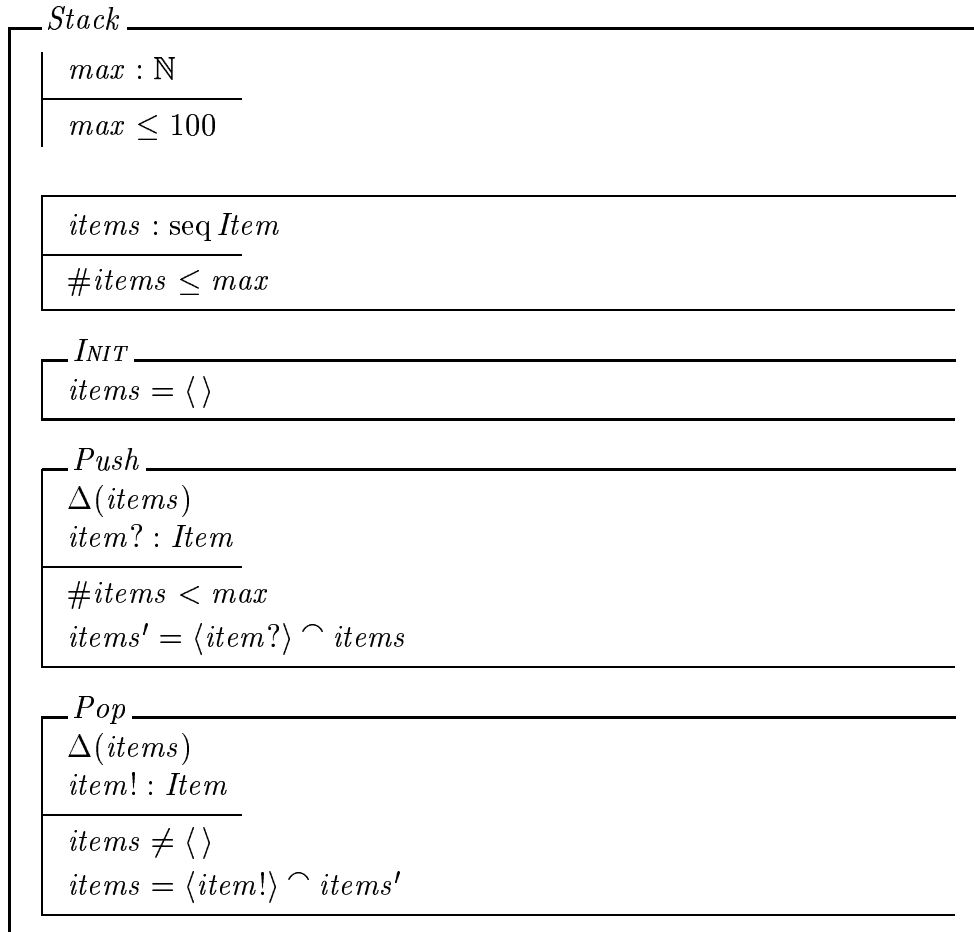


Figure 3.9: A class for stacks in Object-Z