

Part I

Background

Chapter 1

Introduction

1.1 Software reuse

Traditional software development practices have typically been rather ad hoc, with software being developed from scratch each time. However the emergence of the so-called *software crisis*, with the release of software over-budget and well past the due delivery date being the norm rather than the exception, led to a rethink of these development practices. The idea was to bring engineering practices into software development.

One method that was developed in an effort to tackle the software crisis is *software reuse* [53, 11], in which pre-existing software is used to develop new software. A big potential advantage of reusing software is the decrease in development time, which can lead to an earlier release date, as well as a decrease in the cost of development.

Another potential advantage is the greater assurance that reusing software can bring to the overall development. This conjecture relies on the principle that pre-existing software will have already been tested and used to some degree, and indeed the more it is reused the more it has been tested. This can lead to a product that is more reliable and as a result requiring less maintenance later in the product's life. In the case of safety or security critical applications there will be less chance of failures leading to undesirable consequences and costly litigations.

Software reuse is slowly becoming more widespread with a number of successes reported [75, 22, 56]. The NATO Standard for the Development of Reusable Software Components [68] cites a number of industry examples where software has been reused with great success. Biggerstaff [10] cites a reuse case-study by Hewlett-Packard's instrumentation group in which increases in the amount of software reuse resulted in significant decreases in the time-to-market. The paper also notes that there were considerable reductions in the number of defects detected in code developed using reuse in comparison to similar projects by the same group.

So why is software reuse not more widespread? One answer might lie in the fact industry has a tendency to take many years to pick up on research innovations, preferring instead to stick to the old tried and tested ways. However, while this might account for part of the problem, there are deeper issues involved, which can be broadly divided into business related and technical. While business related issues are as much an impediment to software reuse as the technical issues [24], they are outside of the scope of this thesis and will not be discussed further.

In tackling the technical issues relating to software reuse, one should be guided by the doctrine which states that "reusing software can only be attractive to the reuser if the overall effort in reusing the software is sufficiently less than the effort in developing the same software, of an equal quality, from scratch" [77]. That is, the paradigm supporting software reuse must not only make reuse possible, it must also make it cost-effective.

There are two main paradigms for supporting reuse - *composition-based* and *generation-based* [11]. In composition-based technologies, the entities of reuse, referred to as *components*, are self-contained, and can include subroutine libraries, routines, data structures, programs, objects, etc. Reuse is accomplished by the user by finding and adapting components to suit their application. Generation-based technologies [10] are based on the user giving directives which lead to the generation of software. An example of a generative technology is program transformation, where the user gives a specification of their program, which is then transformed via a number of interactive steps to code.

The focus of this thesis is composition based technologies, however it should be noted that there is normally a degree of cross-over between the two. Indeed, approaches to software reuse are seldom purely composition-based or purely generation-based.

1.2 Composition-based reuse

The idea of constructing programs from *reusable software components* was introduced by McIlroy in 1969 [65]. The idea is analogous to building a complex electronic device from a number of smaller, simpler, well-known components in an electronic engineering context. The engineer browses a catalogue of component descriptions for suitable components which can be pieced together in some manner to build their device. To build their device it may be necessary to modify the catalogue components in some way.

A *reusable software component* is defined in [68] as “a software entity intended for reuse; may be design, code or other product of software development” (see also [52, 13, 82] for more details). As well as source and object code, the scope of reusable components can include other software artifacts such as requirements, designs, specifications, tests and documentations [54, 17]. Composition-based reuse approaches have steadily evolved over the years as described below.

An early approach is to use *library subroutines* (such as those used in the standard C library [51]), to develop part of a program. For example, a numerical subroutine library in Fortran may provide a collection of functions for performing certain computations. This may result in reductions in programming time for the developer. However little support is offered to the user in using the subroutine; the user will in general have to view the code to understand what the subroutine does.

In an attempt to address some of these problems, *module*-like structures were introduced in programming languages, such as *packages* in Ada [28]. Ada *packages* offer support for reuse by supporting information hiding and data abstraction [23]. Both of these mechanisms allow the user to get an idea of how to use the package

without having to look at the underlying code. Also, by defining Ada *generics*, packages can become more adaptable, by allowing packages to be parameterised over data structures. However the focus of these approaches is still on the code level.

Next the idea of object orientation was introduced, which supports software reuse [8] throughout various stages in the software life-cycle, including: formal specification (e.g. using Object-Z [15]); software design (Booch [12]); and coding (e.g. using languages such as C++ [90]). Object orientation aids reuse by offering support for abstraction, information hiding, inheritance and class hierarchies. However, one of the main failings of object orientation is the lack of interoperability between classes implemented in different programming languages, or indeed the differing behaviour of compilers [76], meaning that components can only be reused within a certain environment.

As a means of bringing together the strong points of these and other approaches, as well as addressing some of their shortcomings, the NATO standard for developing and using reusable components [68] was developed. The issues it suggests should be considered when designing reusable software components include:

- The designer should minimise a component's machine and implementation language dependencies; any dependencies should be isolated and well documented.
- Each Reusable Software Component (RSC) should implement a single, complete self-contained object. The user of an RSC should not have to extend the functionality of a component.
- Top-level characteristics of a component should be clearly separated from implementation-specific details. Importantly, each component should have an *interface specification* which captures the essential properties of the component (more details below).
- A trade-off between the generality of a component and its simplicity should

be made. In particular very general components are not very useful if their interface becomes too complicated for the user to understand and use.

Components which follow the above design principles are a key factor in making the component-based development paradigm successful. The other key requirement is a framework which supports the component-based development paradigm by providing suitable methods and tools. Clearly such a framework needs to support *adaptation* and *retrieval* of components.

It is important that users be able to make small modifications to components to suit their problem; without this the system is too rigid and may force the user to conform to a set way of doing things, or worse still not provide a solution to the user's problem. The *adaptability* of a component is a measure of how easy it is to modify a component to solve a particular problem.

Retrievability is a measure of how easy it is for the user to find a suitable component amongst a library of components. This becomes particularly important once the library of reusable components becomes quite large. Adaptation and retrieval are described in more detail in Sections 1.4 and 1.5 respectively.

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 shown to terminate.

Such properties can add value to a component, but establishing such properties will often account for a major part of component design. For this reasons it is very important that the framework for using reusable components preserves any properties associated with the components.

1.3 Component interfaces

As a means of supporting information hiding and abstraction, each reusable component should have an *interface specification* (in this thesis the terms *interface* and

specification will both be used in place of *interface specification*), describing the essential properties of the component. The interface defines how the component is used within the reuse framework, while abstracting away many of the lower level details. It provides a description of when the component can be reused and how it can be reused. Interfaces are expressed using either an informal or a formal language. Such interfaces are referred to as *informal interfaces* and *formal interfaces* respectively. In some cases a component and its interface are the same, particularly when the component is itself expressed using a formal language.

Informal interfaces are text-based and typically either consist of a number of attributes [14] or an unstructured textual description [27]. Attribute-based component interfaces are normally searched using a keyword-based search mechanism [14]. For these kinds of systems the most important part is assigning attributes to a component in a systematic manner [77, 4]. Components with unstructured textual interface are often searched using an information retrieval system [4, 27]. For the remainder of this thesis the focus will be on components with formal interfaces.

Formal interface specifications are precise, machine-interpretable descriptions of component's behaviour [78, 92]. The formal interfaces of components used in programming languages take many forms, reflecting their different uses. One example is *functions* in functional programming languages such as ML [84]. The interface of functions in ML is usually considered to be the *function signature*, however since the semantics in ML have a strong mathematical foundation the interface could also be considered to be the entire function. In the Ada programming language [7], an example of reusable components is *packages* whose interface consists of declarations for data structures, functions, procedures etc. Another example of components with formal interfaces is *classes* in C++ [90]; the interface in this case consists of declarations of data types and functions.

Other examples of components with formal interfaces include: theories, composed of axioms, definitions, theorems, etc., in theorem provers such as Isabelle [73] and HOL [1]; state and operational schemas, and axiomatic definitions in Z [87]; functional statements in KIDS [83, 88]; abstract state machines in B [55]; processes

in process-oriented languages such as CSP [18]; classes in Object-Z [21]; and combinations of the above constructs in wide-spectrum languages, such as RAISE [70] which includes functions, state, operations and processes.

There a number of advantages of using a formal language for expressing component interfaces [85]:

- The expressiveness of formal languages and in particular the ability to express concepts with precision and with less ambiguity than can be done informally.
- Formal languages can be used as the basis for building automated knowledge-based tools, which take advantage of the machine-interpretable nature of formal languages.
- Formal languages can be studied and analysed using mathematical methods (referred to commonly as *formal methods*).
- It may be possible to establish the correctness of a component's implementation (with respect to the specification), if the implementation language has a formal semantics [83, 59, 55].
- Formally expressed interfaces can be used as the basis for establishing test cases [34].

The last two points in particular suggest that components with formal interfaces offer better support for validation and verification (V&V), which can lead to greater levels of assurance. By applying suitable V&V techniques to components prior to them being added to the library, users of such components can have more confidence in their correctness and reliability.

There are also some disadvantages of formal specification versus informal specifications which should be noted [85], in particular:

- Yet another language to learn.
- Need for software engineers to have a sound background in mathematics. The development and understanding of formal specifications requires familiarity with discrete mathematics and logic.

- Much of the focus of research on formal specification languages has been on notations and techniques. As a result there is an inherent lack of tool support.
- Difficulty in expressing some problems formally, particularly interactive components of user interfaces, and some classes of parallel processes.

With these disadvantages in mind, it is important to temper the expressiveness and power of formal languages with the need to keep the interface relatively simple. This point is in keeping with the fourth point listed in regard to designing components in Section 1.2 above. The choice of interface (and interface description language) depends on many factors, however in all cases it is important that these choices are guided by the simplicity of the overall approach.

Much of the literature on reusing formally specified components focuses on reuse of singular components, referred to here as *units*. Examples of units include: functions and procedures in imperative languages; schemas and axiom definitions in Z; theorems, definitions and operator declarations used by theorem provers; and functions in declarative languages like ML.

However, as Pree [76] argues, it is often useful to package together related collections of units. For example we might want to group together a number of units which collectively implement some protocol or perhaps a collection of units for creating and manipulating a particular target language data structure. Collections of units are referred to here as *modules*.

Many traditional programming languages provide support for modules; for example C++ provides classes and templates, while Ada provides packages. Similarly a number of formal specification languages provide support for modules : e.g. Sum [50] adds module structuring to Z. In Object-Z [15] the modules are called *classes*, which can contain a number of different units such as schemas, axioms definitions etc., together with sub classes. Most theorem-proving systems provide module-like structures called *theories*, which consist of a number of individual units such as theorems, axioms, constants and definitions [58].

1.4 Adapting components

The adaptability of a component is a measure of how easy it is to modify the component to suit the user's requirements. The level of adaptability is determined by the generality of the component, as well as the methods and tools provided by the reuse framework for applying adaptations.

One advantage of adaptation is that a library of adaptable components can solve a much wider range of problems than a similar-sized library of rigid components. The library of adaptable components is obviously far more attractive to the user than the rigid library, since it is more likely to contain a component which solves a particular problem.

From another point of view, to provide a library with components sufficient to solve a certain set of problems, the rigid library will in general be much larger than the adaptable library. The advantage of having a library of adaptable components in this case is the reduction in the search space when searching for suitable components (whether this is a manual search, or a tool-assisted search). That is it will generally be easier to find suitable components within the library of adaptable components, than in a library of rigid components.

A further advantage of adaptation concerns the design of components. While more effort will be required to design single adaptable components, the designer will be required to design far fewer components. Indeed the designer will be able cover a multitude of problems within a single adaptable component, which would be infeasible and in many cases impossible to cover by rigid components.

The use of *parameters* in components is one particular mechanism for adapting components in a systematic well-defined manner. For example parameterisation has been successfully used in functional programming languages, such as ML, where functions can be passed as parameters and returned as results of other functions. By assigning different values to parameters (*instantiating* the parameter), one component can be adapted to solve a range of problems.

Parameterisation is a useful technique in respect to property preservation. If a

property holds for a parameterised component, it will usually hold for any instances of that component. It is also possible to establish properties for certain parameter values. In this case, to show that the property has been preserved, it is a matter of showing that the parameters have been instantiated to valid values.

Volpano and Kieburtz [91] describe “software templates” which are defined over values of polymorphic abstract data types. An advantage of this is that decisions on how the data is represented can be delayed until later in the development process by allowing types to remain parameterised until it becomes more evident how the data should be represented. This means that a given template can be reused with a number of different underlying data types, making it a more viable entity for reuse.

Goguen [29, 30] extends the scope of component parameterisation further using “module expressions” which provide a framework for performing a number of modifications to components, including: extending a component’s functionality, restricting some of the component’s functionality, combining two or more components and modifying code within a component. All of these modifications can be accomplished while still preserving selected properties of the component expressed as theories in the component.

1.5 Retrieving components

Retrievability is a measure of how easy or difficult it is to find a component in a library which meets the requirements of the user. The component-based development paradigm relies on a large library which will typically be growing in size as more algorithms and data structures etc. are developed. As the library gets larger the difficulty for the developer in finding appropriate components also increases. To improve the level of retrievability and to make the component-based paradigm more practicable, appropriate tool support for retrieval must be developed [66, 45].

Retrieval tools are modelled here as a function which maps a *search query* and a *library* to a set of components (referred to as the *patterns*). The search query encapsulates requirements, selected by the user, that components from the library should

satisfy. The library is modelled as a set of components (extra information pertaining to how the library is structured e.g. component hierarchy, inter-dependencies etc. will not be considered here). The set of components returned by the retrieval tool are those components from the library which satisfy the requirements in the query. The search is based on finding library components whose specification satisfies or *matches* the search query in some sense.

Retrieval tools can be divided into four categories - classification (or faceted) based retrieval, information retrieval, signature-matching and specification matching. The first two retrieval strategies generally apply to components with informal interfaces, and have been discussed briefly in Section 1.3. Signature-matching is discussed in Section 1.5.1 below. Specification matching is discussed in Section 1.5.2.

Note that while these different retrieval strategies have been treated separately, it is possible to unify multiple strategies into a single framework [5].

1.5.1 Signature matching

Functions in functional programming languages like ML have a *signature* which describes the types and numbers of inputs and outputs of a function. A library of such functions could therefore be searched by using the signatures as the search query. More precisely, the library is searched by giving the signature of a desired component (the query) and attempting to match this signature against the signatures of the library components (the patterns). Such a retrieval technique is often referred to as *signature matching* [96, 79, 81].

Rittri [79] describes how signatures can be used to search a library of functions (in the context of a functional programming language), by retrieving information about all functions with this or an *equivalent* type. Runciman and Toyn [81] present a similar approach, noting that using a polymorphic type system for a functional language offers flexibility in combining components, while at the same time limiting the combinations to those which are sensible.

1.5.2 Specification matching

A more general approach to signature matching, is *specification matching* [97]. Retrieval is based on the user describing properties of the desired component in the form of a formal component specification (the search query). A search is conducted by matching the query against the specifications of library components. Specification in this context refers to a general class of formal interface specifications.

Rollins and Wing [80] investigate specification matching on units with Larch-like [31] pre- and post-condition specifications. The retrieval system is implemented in λ Prolog, a “higher-order Prolog”, and as such the built-in unification can be exploited to implement the specification matching. The authors note however that one limitation of this approach is that logically equivalent specifications will often not be matched under this system. For example, a component with a post-condition stating a list is empty ($Iempty(x)$) won't match with one with a post-condition stating that the length of the list is zero ($length(x) = 0$).

A number of systems are available that perform specification matching at the level of units with matching up to logical equivalence. The Inscape system [74] uses the Inquire predicate-based search mechanism to aid the user in the search for reusable components such as operations, data objects and modules specified using preconditions, postconditions and obligations. The Inquire search mechanism can look for predicates that are equivalent to the query predicate, as well as predicates that are weaker or stronger. The VCR system [25] uses implicit VDM specifications as queries for retrieval of software components. The search mechanism in this case searches for components with weaker preconditions and stronger postconditions.

Another retrieval system which uses specification matching is the AMPHION system [89, 62, 63], which makes use of a library of formally-specified FORTRAN routines. AMPHION converts space scientists' graphical specifications into mathematical theorems and uses automated deduction to try to construct and verify a program that satisfies the specification. The success of AMPHION in its particular problem domain is further evidence that reuse of library routines can be made effective with appropriate tool support.

Zaremski and Wing [97, 95] give details of a variety of different equivalences for matching units with pre- and postcondition specifications, giving a comparison of these different methods applied to a variety of problems. These equivalences include: exact pre/post where the corresponding preconditions and postconditions are equivalent; plug-in match where the precondition of the library component is weaker than that of the query, and the postcondition of the library component is stronger; and exact predicate where the conjunction of the precondition and postcondition for the query and library component are equivalent.

Zaremski and Wing also extend specification matching to the module level. For module matching the user gives a set of type and function specifications as a query. The query matches a module if each query type specification matches a type specification from the module and each query function specification matches a function specification from the module.

1.6 Integrating adaptation and retrieval

By integrating adaptation and retrieval, retrieval can be used not only to find 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. By describing successful matches in terms of an adaptation of the pattern, the user does not have to provide the input for performing adaptations. By mechanising the adaptation process in this way, the chances of errors caused by the user giving an incorrect adaptation is lessened.

While the specification matching approaches mentioned so far provide good solutions to the retrieval problem their support for adaptation is restricted to parameterisation. Jeng and Cheng [46] however extend adaptation within the retrieval framework. The library components used in this approach have a specification part and a program part. The retrieval process is divided into three separate stages. In the first stage, the user gives the specification of their desired component (the query), and specification matching is performed to find the set of matches. In the

second stage, called program replacement, for a match with a given library component, the program for that component is refined to an intermediate program by renaming variables, renaming identifiers and instantiating parameters, according to the match information. In the third stage, called program adaptation, the intermediate program is modified to give a program which satisfies the query specification. While the first two stages are automatable, the third stage will require a degree of user interaction.

1.7 This thesis's approach to component reuse

1.7.1 Overview of solution

This thesis presents a generic framework for using reusable software components. The focus is on components with formal interface specifications for the reasons discussed in Section 1.3. The framework includes support for *adaptation* and *retrieval* of components.

The general framework for adapting and retrieving components is defined in this thesis by partitioning the language features, which appear in the formal languages used to represent component interfaces, into three separate levels (see Fig 1.1). Therefore it possible to develop techniques for adapting and retrieving component relevant to each level, which is important since different techniques apply at the different levels. At the lowest level are *mathematical expressions*. The next level of components will be referred to here as *units*. The third level will be referred to here as *modules*. Solutions to adaptation and retrieval are then developed separately for each of these three levels of formal language constructs.

Mathematical expressions are the building blocks for formal component specifications. Many different mathematical formalisms are in use in software development. Examples include propositional logic, first-order predicate calculus, typed lambda calculus, Zermelo-Franko set theory etc.

Units come in many forms: e.g. functions in applicative high order programming languages such as ML, where the specification is a signature; definitions, axioms and

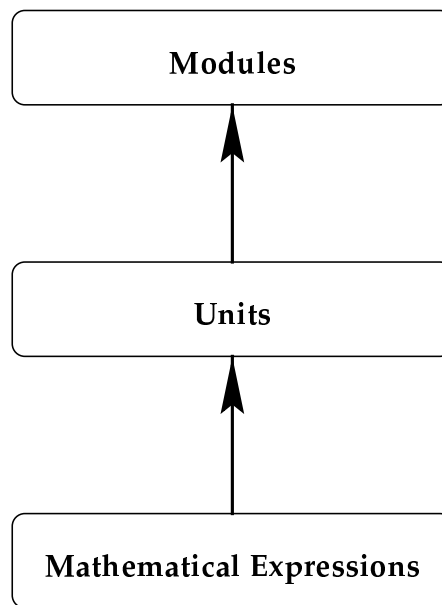


Figure 1.1: A 3-tiered classification scheme for reusable software components

theorems in theorem-proving systems whose specifications may be a signature or the entire construct [58]; function-like statements in formal software development environments, where the specifications may be a signature, or a pre- and post-condition specification [83, 49]; state and operational schemas in Z , where the specification may be the signatures of the state variable, or the entire construct [87]; and processes in process-oriented languages such as CSP [18].

The first part of the solution is a general framework for adapting components, containing several common component adaptation techniques. The thesis proposes developing this framework by looking at techniques relevant to each of the three levels of specification constructs separately (see Table 1.1). At the expression level the adaptation technique explored is parameterisation. At the unit level, identifier renaming, renaming of variables and reordering of the inputs and outputs of function-like units are explored. The main adaptation technique at the module level is restricting modules to self-contained subsets.

The second part of the solution is a framework for retrieval of formally specified components which mirrors the approach to component adaptation. The solution to retrieval is based on *specification matching*, however in this case the notion of

Component Level	Adaptation Techniques
Expression	Parameter instantiation
Unit	Identifier renaming Variable renaming Argument reordering
Module	Subsetting

Table 1.1: Component adaptation techniques partitioned into three levels

specification is much broader than other approaches to specification matching. The specification of the desired component (the *query*) is matched against the specification of a library component (the *pattern*). Matches between pattern and query are described in terms of adaptations of the pattern required to give a specification equivalent to the query. Solutions to specification matching correspond to the adaptation approaches for each of the three levels of components.

As a means of testing the general ideas in a more concrete and formal sense, the adaptation and retrieval techniques were integrated with the CARE tools. CARE was chosen because it is relatively small and simple, yet covers many features common to other formal languages. In particular, CARE includes: variables; functions; predicates; binders; parameters; inputs and outputs; typing; pre- and post-conditions; separation of specification and implementation; and modularity. Each of these common features is exploited in some sense in this thesis to support reuse (and in particular to support adaptation and retrieval).

Next a tool for semi-automating parts of the development process in CARE was built. This tool makes use of the adaptation and retrieval tools, and assists the user in finding and adapting suitable library components to perform refinement steps, based on the current state of their development. To illustrate the effectiveness of these enhancements to the CARE toolset, and the value they add to the overall CARE methodology, several small examples are developed using the CARE tools.

Two papers, which report on the ideas presented in this thesis have been published [36, 41]. A technical report has also been prepared, which reports on the approach to module matching [38].

1.7.2 Key contributions

The overall solution is general, and applicable to a wide range of formal languages. The solution provides a framework for using components which is simple for the software engineer to use. The solution provides a number of commonly applicable techniques for adapting and retrieving components, which collectively greatly enhance the value of a library of reusable software components.

The key contributions of this thesis are as follows:

1. A general framework for adapting formally specified components, consisting of a number of adaptation techniques, based on the component interfaces. These techniques are relatively simple, making them fairly easy for the user to understand and use. The techniques are also very general, covering many of the common kinds of language features found in formal specification languages, from a broad range of development methodologies.
2. A framework for retrieving components, based on the user providing requirements of a desired component, in the form of a search query, and then retrieval tools searching the library for components which match the search query in some sense. This approach generalises other specification matching approaches, since it allows for a wider range of formal specifications to be used.
3. A unified framework for adapting and retrieving components. Matches between the user's search query and library components are described in terms of adaptations to the library component. This means that the user is given support in getting adaptations right, thus avoiding the situation of the user entering an incorrect adaptation, which may not be discovered until sometime later in the development process.
4. Classification of components and component interfaces languages into a number of separate tiers: three tiers are suggested in this thesis. Classifying components in this manner means that issues pertaining to certain kinds of components can be more easily isolated, leading to a more general approach. This

layering approach also means that changes to the adaptation or retrieval mechanisms for components in a particular level can be done without affecting the overall mechanism at the higher-levels. For example the adaptation mechanism could be modified at the unit level with minimal changes required at the module level.

5. Development of a generic retrieval tool (referred to as the *search engine*) which is configurable to the needs of the user and/or application. Such a tool can easily be integrated with existing development tools without requiring major changes to the existing tools. This tool also allows the user the flexibility of choosing between the precision of a search, and the efficiency.
6. Extension of the CARE methodology and toolset with tools for retrieving and adapting the reusable components in CARE.

1.7.3 Outline of thesis

The solution is presented in this thesis is four parts: an introduction; a solution to adaptation; a solution to retrieval; and applications of the overall approach.

The CARE language is described in Chapter 2 including a formal specification of the abstract syntax for relevant parts of the language. On the first read this chapter could be skimmed by the reader, with a more detailed read done as required.

Chapter 3 describes a general framework for adapting reusable components. A variety of component languages are explored, from which a number of simple, widely applicable adaptation techniques are derived. The techniques are developed in the CARE language in Chapter 4, by looking at the three levels of components (expressions, units and modules) separately.

In Chapter 5 a general framework for component retrieval is described, based on using specification matching. The approach to retrieval is integrated with component adaptation by describing matches in terms of adaptations to the library components. The examples introduced in Chapter 3 are revisited here in the context of retrieval.

Chapter 6 contains algorithms for matching mathematical expressions in CARE. The chapter firstly contains a specification and design of a basic algorithm for matching expressions. The remainder of the chapter describes a number of enhancements to the basic algorithm, including AC-matching, interactive matching, type-constrained matching and unification.

Chapter 7 extends the retrieval approach to units. Unit matching algorithms are given for each of the different kinds of units in CARE. Each algorithm for matching two units is based on matching the corresponding subunits of the two units.

An approach to matching modules is developed in Chapter 8. A number of different matching strategies are described, leading to an overall approach which is quite flexible. The final part of the chapter describes how the matching algorithms can be used to build a generic search engine, which will form the basis for building retrieval tools.

In Chapter 9, the general search tool described in the previous section is adapted to build retrieval support into an existing set of tools in CARE, thus enabling certain steps in the development of CARE programs to be performed semi-automatically. A number of examples are also given which illustrate these retrieval tools, as well as illustrating many of the other ideas in the thesis.

Chapter 10 summarises the findings of this thesis, as well as noting some of the further work that could be done to improve on the overall approach.

Appendix A defines the predefined theory in CARE used in this thesis which forms the foundation for the specifications of CARE components. Appendix C contains the library templates used in examples in this thesis. At the back of the thesis, an index of definitions and concepts is provided for easier reference.