

Chapter 10

Conclusions

10.1 Summary

The so-called software crisis forced a rethink of software development practices a number of years ago. One idea to come out of this was the idea of reusing software during the development. One of the main paradigms for software reuse is composition-based reuse, in which software is constructed from reusable components.

Two main issues must be considered in developing an approach to reusing components. Firstly to avoid a combinatorial explosion of the size of the library of reusable components, components should be adaptable, meaning that only one version of a component is required rather than many variants. Secondly, to help the user find suitable components for their problem, support for component retrieval must be offered. As the library increases in size, structuring of the library is no longer sufficient and indeed becomes quite difficult, thus tool support is required. Furthermore these approaches should be tightly coupled to gain the full benefits of both.

This thesis presents an approach to reusing formally specified software components which offers an integrated solution to the problems of adaptability and retrievability. The fact that the components are formally specified is exploited in developing algorithmic solutions to both problems. The notion of what constitutes

a component specification is very general throughout this thesis, as is the notion of a component, which includes components at several different levels of granularity. The framework for adapting and retrieving components has been formally specified and illustrated in depth (on CARE). Furthermore it has been prototyped as tool extensions to the CARE toolset and validated on a number of case studies.

Chapter 1 introduces the problems associated with software reuse. Chapter 2 summarises the CARE language, used to explore the ideas presented in this thesis. Chapters 3 and 4 develop a framework for adapting components, the first of these chapters giving a general proposal, the second chapter implementing the ideas in more detail in CARE. Chapter 5 introduces a general framework for matching and retrieving components. Component matching is developed in more detail for CARE in Chapters 6-8. Chapter 9 describes the development of a library retrieval tool for CARE, which is then illustrated with a number of development examples.

The following subsections describe briefly the key contributions that the thesis makes towards developing an effective approach to reusing software components.

10.1.1 Support for adaptation

This thesis presents a general framework for adapting formally specified components. The framework consists 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 general, since they cover many of the main kinds of language features found in formal specification languages, from a broad range of development methodologies.

The simplicity of these techniques also means that tools, which perform these adaptations automatically, can easily be developed and integrated with existing development tools, with little or no change to the existing tools and development methodology required. Extending the approach by adding new adaptation techniques should be relatively straightforward within the general framework.

10.1.2 Support for retrieval

Also presented in this thesis is a framework for retrieving reusable components. The approach is 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. The search query is matched against the specifications of library components based on a technique called *specification matching*. Using formal specifications as the basis of component retrieval means that the user can express more sophisticated and precise requirements in the search query.

10.1.3 Integration of retrieval and adaptation

The approach to component retrieval presented in this thesis has been integrated with adaptation. 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.

Basing retrieval on formal specifications of components is a more promising approach than more traditional approaches. For example consider the `accumulator` template (see Fig. C.1) and the `if then intro` template (see Fig. C.3), which are both fairly general. These templates would be difficult to classify, because they don't relate to any particular data structure, therefore making traditional keyword and classification based retrieval mechanisms less effective. However the integration of adaptation with a (formal) specification matching based retrieval tool makes it easier for the user to retrieve these general components, since this approach does not rely so heavily on classifying the components.

10.1.4 Generality of retrieval tools

The retrieval tool developed in this thesis (referred to as the *search engine*) has been developed in such a way that it is general and it is configurable to the needs of the user and/or application.

The generality of the search engine means that it can easily be integrated with existing development tools without requiring major changes to the existing tools. It also means that the search engine is adaptable to fit in with different development methodologies.

The search engine allows the user the flexibility of choosing between the precision of a search, and the efficiency. This is supported in a number of ways, including: allowing choice between expression matching equivalences; the ability to enable or disable techniques for narrowing the search space (such as type constrained matching and interactive matching); and a choice between a number of module matching strategies.

10.1.5 Classification of components

Another finding from this thesis is that components can usefully be classified into a number of separate tiers: three tiers are suggested in this thesis. The bottom tier contains the most basic entities of reuse, such as mathematical expressions. The second tier contains single stand-alone units, and the third tier contains module like structures which are made up of a collection of individual units. 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 matching 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.

10.2 Further Work

A number of suggestions for further work follow; the points described below can be categorised as either experimental analysis of the tools or extensions to the approach.

10.2.1 Scaling up

The approach presented in this thesis has only been tested on a relatively small sized library of small components (<100 components, each contain <100 LOC). It is generally considered that approaches to software reuse which work in the small scale do not necessarily work in the large scale. This would suggest that a study of this approach in the larger scale is required.

While it is not desirable to apply this approach to monolithic components, it is reasonable to expect that this approach to component reuse should be able to cope with libraries containing 1000's of components, each with something in the order of 1000 LOC. For example the AMPHION tool has been successful in handling large libraries, while Mural's theorem library numbers in the multi-thousands.

A study of the approaches to scalability would involve several steps. Firstly more work would be required on the prototype tools, concentrating on their stability, efficiency and user friendliness, thus bringing the tools more towards industrial usability. Secondly, substantial population of the library would be required, which would involve the development of both general as well as domain specific components. Finally an industrial strength case study(ies) should be developed, using metrics to measure the success of reuse versus that of more traditional development methods (see next section for more details on metrics).

10.2.2 Metrics for reuse

The key principle of software reuse is that the overall effort/cost of developing software using reusable components must be significantly less than the effort in developing the software from scratch. It is hypothesised in this thesis that an approach to component reuse which supports adaptation and retrieval, using formally specified

components will reduce the effort of reuse. However no measurements have been done to ascertain exactly how effectively the approach achieves its goal of meeting the above principle.

A study of the effectiveness of the approach could be undertaken by introducing some metrics into the aforementioned case study. For example the overall cost/time/effort could be measured for a development which reuses components versus a development done from scratch [26]. Such measurements may take into account the time taken to populate the library, particularly with domain specific components. Such a study should also take into account maintainability and other “full-lifecycle” costs, therefore giving a measure of the benefits of reuse on the entire life of the software.

Another useful metric would be the actual percentage of software reused in the development. Metrics could also measure the number of bugs reported for the software developed from reusable components, versus that developed from scratch.

10.2.3 Existing component libraries

Another possible extension is the idea of integrating the component reuse approach described in this thesis with existing component libraries, and in particular investigating the issues involved in doing so.

Components in existing libraries may or may not be formally specified, it can be assumed though that the majority of components are not. Part of the integration process will then be adding formal specifications to the component interfaces.

Formally specifying existing components would in general be a difficult and time consuming task, particularly if the component’s body must remain unchanged. This means that the specification must be retro-fitted to the code, which is generally much more difficult than giving a specification up-front and developing the code from the specification. The process of formally specifying existing components would involve: familiarisation with the components; specification of the component; validating and verifying the component.

It would be an interesting study to see whether “enhancing” existing components

in this way is a productive exercise, or whether the components are best developed from scratch, starting with the specification, then developing the code (or other kind of software). This question parallels the general principle of software reuse.

10.2.4 Investigating other languages

While a concerted effort has been made in this thesis to ensure that the approach is general by looking at a variety of languages, the ideas have only been fully investigated within the framework of the CARE language. Due to the author's more detailed and perhaps biased knowledge of the CARE language, when compared to a less in-depth knowledge of other languages, it is conceded that certain issues may have been overlooked.

To overcome this potential shortcoming, it would perhaps be desirable to look at other languages in more depth. In particular, to overcome the bias of CARE towards model-oriented, and in particular functional specifications, it would be a good idea to look at state based specification languages, process-oriented languages and object-oriented specifications languages in more detail in an effort to further generalise the approach.

Another feature not considered in depth in this thesis is hierarchically structured module languages. The approach explored in this thesis has used a "flat" library, i.e, all library components are stored in a single directory. For large libraries some structuring of library components is generally required, using some kind of classification schema. This may introduce new complexities to the overall problem, so a study which investigates this problem of integrating specification based retrieval and component classification would be worthwhile.

The main focus of this thesis, with respect to adaptation, has been at the specification level, with only a brief discussion of adapting components at the code level. The proposed solution to target language adaptation is to link the adaptable parts of the code to the specification in some manner, so that adapting the specification results in an adaptation to the code. However this proposed solution has only been fully tested on one particular application (polymorphic data structures) on one lan-

guage (C). That is, by instantiating a type in the specification, a data structure used in the code can be changed. A more detailed study of different ways of adapting the underlying target code, looking at a variety of target languages is required.

10.2.5 Other applications of the search engine

The search engine has only been applied to one particular application (a retrieval tool) in one development language (CARE). Development of other applications using the search engine would be a useful study. One such application might be in the development of a *library browser* tool.

A *library browser* tool would use the search engine quite differently. The search query would be created by the user, rather than coming from the current program (although the user might cut-and-paste it from a program). In this case the user would be free to include parameters in their query, enabling them to under-specify their query in some sense. The library browser would not impose any restrictions on the results returned, in particular they don't need to be consistent in any manner with other units. This means that the library browser allows the user to select any of the available search methods, unlike the program development tool which restricted the user to correctness preserving search methods. Like the program development tool, the library browser tool allows the user to enable/disable a number of options, as well as select an interaction level.

The output of the search engine will also be treated differently by the library browser. The user might be interested more in the modules matched, rather than the units that result from applying module adaptations; or they may be interested in the actual adaptation. The library browser would have switches for changing the view of the results returned. Like an internet search tool, the library browser might also include a feature for sorting the search results in some manner.

Finally the library browser tool might incorporate a traditional keyword-based retrieval strategy with the specification-based search engine. A keyword-based search could be used first to reduce the search space by eliminating unsuitable modules or by providing the user with non-formal component details such as perfor-

mance characteristics or development history. The specification-based search engine would then be applied to the remaining modules in the manner described above.

10.2.6 Completeness of algorithms

While many of the matching algorithms presented in this thesis have been shown (informally) to be complete, some algorithms are known to be incomplete. In particular, the unification algorithm used in the prototype tools is a very basic one, which is sufficient to test the general ideas. However there are many other more sophisticated algorithms, based on Huet's paper on unification of lambda-terms, which capture more unifiers and are possibly more efficient (note that the choice of matching algorithms will depend heavily on the specific language).

There are times when the user might want to be sure that there is no possible match. In this case completeness is a vital issue. To this end, more work on ensuring the completeness of algorithms would be required. Naturally this would restrict matching and unification to a certain subset of expressions, since in general unification is not decidable. This would also require adapting the search engine so that the user can specify whether or not they want to be notified of completeness.

10.2.7 Displaying search results

Another issue not covered in great depth in this thesis is displaying of search results. The current prototype retrieval tool simply displays the results in the order that they were found. A number of enhancements could be made to the system. For example the search results could be displayed in order, with results which "fit the query the best" displayed first. This would require defining an ordering(s) to be used by the retrieval tool.

Also worth investigating are issues relating to how the search results are most usefully presented to the user. Different granularities of information may be useful. At the top level, there may be a summary of the search results, giving a one line description of each match. This summary may include the name of the library

component matched (plus perhaps its location), and a “score”, which indicates to what degree the pattern matches the query. The user could then click on one of the search summary lines to get a more detailed description of that particular match.

10.2.8 Extensions to the CARE tools

The CARE tools could usefully be extended with most of the ideas presented in this section. Extending the tools so that they can handle the pretty printed form of the language constructs would also be useful, since the terse form, while useful for experienced users, is difficult to comprehend for novice users. Also, a tighter coupling of the retrieval tool(s) with the other tools would be useful. Finally, better integration with other knowledge based tools, such as theorem provers and type-checkers is required.