RAL 88052

Copy 1 RG1

RAL-88-052

Science and Engineering Research Council
Rutherford Appleton Laboratory
Chilton DIDCOT Oxon OX11 0QX

RAL-88-052

THE STATE OF THE ART IN SOFTWARE REUSE

A J J Dick, M S Parsons, S C Lambert

June 1988

THE STATE OF THE ART IN SOFTWARE REUSE

A. J. J. Dick

M. S. Parsons

S. C. Lambert

Informatics Dept.
Rutherford Appleton Laboratory
Chilton, Didcot
OXON OX11 0QX

1. Introduction

Reusability is a general engineering principle whose importance derives from the desire to avoid duplication and capture commonality in undertaking classes of inherently similar tasks [Wegner83a].

Most engineering disciplines exploit reusability extensively, to considerable economic advantage, at all levels of the engineering process. The most obvious example is that of physical components, which are usually designed to be reused in a wide variety of applications. Other entities, such as ideas, concepts, specifications, designs, manufacturing techniques and test methods, are reused just as frequently, but in a less obvious way.

The potential for reusability in all phases of software engineering is great. Indeed, a considerable amount of reuse occurs already in the software industry, preventing the duplication of effort, and increasing software productivity.

In this report, we review existing techniques for the promotion of software reuse, discuss their limitations, and suggest possible research directions for the future.

The material presents the results of a literature search in software reuse conducted during the preparation of an ESPRIT project proposal.

2. Existing Techniques that Promote Software Reuse.

2.1. High-level Programming Languages.

Early advances in programming productivity were achieved through the use of high level programming languages. They enforce reuse of very small-scale components, i.e. language constructs, which may be composed in well-defined, highly flexible combinations. With language and compiler standardisation, the resulting code may be used on a wide range of very different machines.

Whilst this kind of reusability is vital, the component level is too small (although the composability of components is excellent), and the reuse of whole programs is limited to portability across hardware.

2.2. Program Libraries

Libraries of software objects have long been used to promote the reuse of code within certain well-defined application domains (e.g. the NAG library for mathematical functions, SPSS for statistics).

Here the individual components are still too small, and many design decisions have been fixed, such as the programming language used, and the number or type of parameters. No provision is made, short of hand-coding, for slightly modifying a component that very nearly matches given requirements.

2.3. Parameterisation.

A programmer who is conscious of the advantages of reusable code is often able to build parameterised software components that are suitable for a wider range of applications than is actually required for current use.

Such a technique is limited by the programmer's imagination, and the fact that some kinds of design decision cannot be parameterised (e.g. the choice of a particular programming language).

2.4. Architectures for Component Composition.

Good combination operators promote the construction of new functions from existing software components. The prime example of this is the provision of pipes in the Unix operating system, in which input and output streams of all programs are, in effect, parameterised, and can be arbitrarily inter-connected by use of an operating system-level language.

Combination operators have proved successful for components that communicate in a well defined way. Combinators for hierarchical composition, in which components may fit inside one another, are much more difficult to define.

2.5. Application Generators.

Application generators and highly configurable packages avoid the need to re-program applications that differ only within the limitations of a particular domain (e.g. report generators, VisiCalc). They promote the reuse of domain-specific concepts at a high level, and many can produce code in a choice of several programming languages.

An application generator can be developed only after the application area is relatively well-understood and standardised – and standardisation cannot keep pace with new applications.

2.6. Meta-programming and Conditional Compilation.

These methods lift the level of parameterisation from run-time to (pre-)compile-time. They typically produce software components by *excision* (removing irrelevant parts of a generic component) rather than by *concision* (constructing a component from smaller parts). A common example of such methods are GENSYS programs, which generate correctly configured operating systems.

Here reuse is limited to the multiple instantiation of a generic component. Usually reuse can occur only in ways which have been pre-determined by the designer. (The greatest amount of added value comes when a component may be reused safely in ways not originally envisaged.)

2.7. The Object-Oriented Paradigm.

Object-oriented programming promotes reuse through inheritance and information hiding. Inheritance allows the programmer to add to the functionality of an existing component in a well-defined way to suit a new need. Information hiding makes the visible part of a component largely independent from the way it is implemented, making the component more reusable at the interface level.

However, limitations are placed on the composability of components; operations in unrelated components cannot usually be combined. And, as with most forms of parameterisation, object-oriented languages imply particular decisions regarding the implementations of data types and operations, limiting the contexts in which components can be reused.

3. Problems and Limitations of Current Reuse.

The majority of existing reusability is at the code level, and thus is of limited value. Despite the fact that reuse is a ''phase-independent'' concept, in as much as it is applicable to all phases of the software life-cycle [Ramamoorthy86a], few current methods support the reuse of designs, specifications and programming methods and concepts.

Futhermore, in the process of constructing software, programmers constantly reuse ideas, concepts and programming knowledge, much of which is catalogued in text books and journals (e.g. ACM algorithms). No existing methodologies promote this level of reuse to any great extent.

One of the most important problems in software reuse is the organisation and retrieval of the available reusable components. A simple ''syntactic'' approach may be adequate for some purposes, but when the range of available components becomes very large or the user's requirement is a complex one, then a semantic approach would be more valuable, for instance in intelligently selecting the ''nearest match'' to the required functionality.

4. State of the Art in Software Reuse and Problems

4.1. Plan Calculus

The Plan Calculus [Rich83a] combines ideas from flowchart schemas, data abstraction, logical formalisms and program transformations. In this system components are represented as *plans* which are stored in a library. The algorithmic parts of plans are represented as hierarchical flowchart schemas, such flowcharts are annotated with logical assertions (expressed in predicate calculus). Each box is annotated with pre- and post-conditions, there are also constraints on undefined parts (*roles*) and a network of dependency links. The basic flowchart representation is extended to contain components which correspond to data and also to sub-computations, and annotations may be added to these parts as well. Plans may be translated into programs and vice-versa and functions between plans are realised by *overlays*. A sound semantics is given to this approach by the use of the situational calculus.

The Plan Calculus supports a layered approach to reasoning: the system can perform simple deductions in terms of predicate calculus assertions, and more complicated deductions in terms of flowchart schemas and overlays. The system is intended to satisfy five conditions on a representation for reusable components. These conditions are: expressiveness, convenience of combination, semantic soundness, machine manipulability and programming language independence. The Plan Calculus is an example of a mixed-mode knowledge representation language, in which different representations are allowed to co-exist because they are suited to different purposes. In this case there is a predicate calculus representation underlying a more complex plan diagram formalism. It seems likely that some kind of mixed-mode representation will be necessary for any representation language for reusable components.

4.2. OBJ

OBJ [Goguen84a] is an executable specification language based on equational logic. A program in OBJ is a hierarchy of modules; such programs are executed by interpreting equations as rewrite rules. As a language it has the desirable properties of referential transparency, separation of logic from control, and the ability to construct proofs and programs in the same language. Its main features are: strong typing, user-definable and parameterizable abstract objects, associative pattern matching, ability to perform program transformations, libraries, theories, views and interactivity. There are also good support tools.

Obj introduces three new ideas: *theories* which declare properties of programs and interfaces, *views* which connect theories with program modules and *images* which can modify existing modules. The image operation allows creation of modules with increased or restricted functionality and also the renaming of the interface. One disadvantage of OBJ is that no help is given in finding modules.

4.3. LIL

The Library Interconnection Language (LIL) [Goguen86a] is a module interconnection language which allows assembly of large programs from existing entities. It supports: information hiding, semantic and syntactic searches of the component library, narrowing of interfaces, flexible binding of generic components, interactive configuration and version management, integration of vertical and horizontal structuring, different levels of formality, and facilities for program understanding. It uses the following ideas: *theories* (which give semantics to software components by providing formal or informal axioms), *views* (which describe semantically correct bindings at component interfaces and hence describe module interconnections), *generic entities* (which give maximal reuse) and both *horizontal* and *vertical* composition.

4.4. TELL/NSL

This system uses an English-like specification language, TELL/NSL [Katoh86a] which is automatically transformed into first order predicate logic. Logic based retrieval of software modules from a module library is supported using a theorem prover to prove functional equivalence of retrieved modules to those required. The theorem prover uses ordered linear resolution plus heuristics. TELL/NSL provides interactive program composition and gives automatic generation of explanations in natural language style. However, the system is not powerful enough to retrieve all modules having specifications equivalent to those required. Also the specifications of auxiliary modules are not examined when determining whether two modules are equivalent. TELL/NSL gives no help to the programmer when deciding whether retrieved modules are fit for the required purpose. Any modifications required must be done manually.

4.5. RECIPE

The RECIPE [Tracz87a] system consists of 3 parts: a component library, a library manager (which employs a Prolog knowledge base to answer queries about components), and an application generator framework for incorporating software components. The RECIPE user interacts with the system to collect the parameters of the application. When this has been done, the system generates an operational program which may be compiled immediately or modified manually. There are two distinct types of RECIPE user: the recipe follower (the application developer) and the recipe creator. The components library has facilities to support cataloguing, searching and retrieval. The main advantage of this system is its interactive nature: menus, online help etc. Two disadvantages of this system are the extra cost of developing interactively parameterized programs and the limitations of the search technique.

4.6. RSL

The RSL (Reusable Software Library) [Burton87a] consists of a database of components and four sub-systems: library management, component retrieval, user query and interactive computer aided design. The database stores attributes of every reusable component (which may be functions, procedures, packages or programs). The classification of components is based on a hierarchical category code and keywords. The library management tools aid the RSL librarian, these include tools to extract reuse information from design or source code files and tools to enter and maintain components in the RSL. User query is interactive and has both natural language and menu interfaces. The *Score* system helps users to evaluate retrieved components. It interactively asks questions about requirements and finally presents an ordered list. It works by comparing information about domains as well as subjective data. Note, however, that the component evaluation is still essentially ''syntactic'', and requires rigidly defined metrics to compare user requirements against available components.

4.7. REUSE

REUSE [Arnold87a] uses a passive approach to reuse: it is a customizable, menu-driven front-end to an Information Retrieval (IR) system. There are 4 predefined component classifications: *template* (the most general), *module* (an entity that performs a single task), *package* (a related collection of modules) and *program*. Each component entry has two parts: a required part in a standard format and a free form part. REUSE uses menus for cataloguing and retrieval of software component information. Each user menu consists of a list of keywords; there is also a system thesaurus to aid retrieval. A system administrator creates and updates user menus and the thesaurus, and also adds new component classifications along with its menus. Users may search for, create, update and delete component entries.

4.8. MELD

MELD [Kaiser87a] blends library, program generator and object-oriented approaches to reusability. Modules are written in a high-level declarative language, these may be translated to an efficient implementation in a conventional language. MELD is like an object-oriented language in that it has encapsulation of data and operations as objects defined by classes. However it is different to such languages because it uses *features* (the reusable building blocks) and *action equations* (which define the relationships between objects and their dynamic interaction). The problems of assessing the suitability of retrieved components and of categorization of components are not addressed.

4.9. FOR-ME-TOO (ESPRIT I Project No. 283)

This project finished in September 1987 after undertaking a limited amount of research into languages for promoting the reusability of components in software systems. For sequential aspects, the generic specification and programming language LPG was studied; for concurrent aspects, Petri nets were investigated.

4.10. ASPIS (ESPRIT I Project No. 401)

The main goal of the ASPIS [Pietri87a] project is to develop and exploit knowledge based techniques in a software development environment, aimed particularly at the most knowledge-intensive phases of the software life-cycle, namely requirements capture/analysis through to design.

The architecture of the environment is based on the use of four knowledge-based tools called ''assistants'', which assist the user in analysing, designing, prototyping and reusing specifications and designs. Specifications are written in a logic-based formalism called Reasoning Support Logic; a prototype of a system is a knowledge base (i.e. a collection of facts and rules describing the system) which the designer ''executes'' by making queries.

The issue of reuse is not the most major consideration in this project. To date, work has not been started on the Resuse Assistant; there is no evidence that the retrieval of reusable specifications will support approximate or near matching when no exact match exists.

4.11. PRACTITIONER/KNOSIS (ESPRIT I Project Nos. 1221 and 974)

The KNOSIS [Cloarec87a] approach to software reusability is a ''programming in the large'' one, where a comprehensive model of software components is being elaborated and implemented using a knowledge representation system. Reuse is supported at the code level (simple components) and at the design level (complex components, or systems of combined components).

The retrieval of components is not by means of classification, but by general search facilities supported by artificial intelligence techniques such as partial pattern matching and unification. Similarities between components will be sought, rather than exact functional matches.

So far, the project has studied a number of reuse problems, including the reuse of test programs and data, and the construction of a complete system from components produced by several development tools. The results of these studies have shown that there is a real need for a methodology of reusability.

5. Knowledge Based Systems in Software Reusability.

It is widely recognised that knowledge based techniques have a part to play in promoting software reuse. The semantic modelling of components is highly desirable [Onuegbe87a] and the established techniques of AI, such as frames and semantics nets, provide one way of making it possible. Some of the systems outlined in the previous sections go a little way towards the incorporation of KBS for component organisation and retrieval. Ramamoorthy et al. [Ramamoorthy86a], while doubtful that an unlimited semantic retrieval will be possible in the foreseeable future, still advocate it for restricted domains or as part of an interactive process involving feedback from the user. A small step in the direction of semantic retrieval has been taken by Woodfield et al [Woodfield86a]. who have classified information about a module into four categories, starting with rudimentary syntactic information and encompassing ''additional comments acquired from an expert specifically designed to aid module retrieval.''

There has been some work on the kinds of representations that designers building systems from reusable components would find useful. For instance, Curtis [Curtis83a] has studied the characteristics of a library of modules that aid in the reuse of components, while Ehrlich and Soloway [Ehrlich84a] propose the use of programming plans represented as stereotypes and rules of programming discourse.

6. Summary of Research Directions.

Methodologies are required that support reuse in all phases of software development, and at all levels of abstraction. The management of components is a complex problem. Retrieval by exact semantic match is obviously not satisfactory; components that could be modified with the least amount of effort to meet functional requirements are also of interest.

The following research directions are particularly important:

- ☐ A general framework and methodology which promotes
    - i) the reuse of programming concepts, specifications, designs and code
    - ii) the design of components for ease of reuse from the outset.
- ☐ Languages for specifying, coding and otherwise describing components, that promote
    - i) the understanding of components
    - ii) the semantic retrieval of components, with particular emphasis on matching the user's requirement as closely as possible when there is no exact fit
    - iii) the description of composability of components, including restrictions on the use of each component
    - iv) the hierarchical and functional classification of components
    - v) the simple modification of existing components.
- ☐ AI and KB techniques for promoting reuse, specifically forms of Knowledge Representation suited to
    - i) the encapsulation and reuse of programming ideas and methodology

ii) the organisation and retrieval of components (again with emphasis on inexact matching)

iii) the encapsulation of knowledge about the composability and limitations of modules

iv) the analysis of test case results (e.g. for the detection of when component limitations have been violated)

v) the management of module development and maintenance.

References

Arnold87a. Susan P. Arnold and Stephen L. Stepoway, *The Reuse System: Cataloging and Retrieval of Reusable Software,* IEEE (1987).

Burton87a. Bruce A. Burton, Rhonda Wienk Aragon, Stephen A. Bailey, Kenneth D. Koehler, and Lauren A. Mayes, ''The Reusable Software Library,'' *IEEE Software (USA)* **4**(4)IEEE, (July 1987).

Cloarec87a. J.-F. Cloarec and R. Valent, ''A Knowledge Based Environment for S/W System Configuration Reusing Components,'' pp. 339-351 in *ESPRIT '87 Achievements and Impact, Procs. 4th Annual ESPRIT Conf., Part 1, Brussels, Sept 28-29 1987,* Commission of the European Communities (1987).

Curtis83a. Bill Curtis, ''Cognitive Issues in Reusability,'' in *Proc. ITT Workshop on Reusability in Programming, 7-9 Sept 1983, Newport, RI,* ITT Programming (1983).

Ehrlich84a. K. Ehrlich and E. Soloway, ''Empirical studies of Programming Languages,'' *IEEE Trans. on Software Engineering* **SE-10**(5)(1984).

Goguen84a. Joseph Goguen, ''Parameterized Programming,'' *IEEE Transactions on Software Engineering* **SE-10**(5) pp. 528-543 IEEE, (September 1984).

Goguen86a. Joseph A. Goguen, ''Reusing and Interconnecting Software Components,'' *IEEE Computer,* pp. 16-28 IEEE, (February 1986).

Kaiser87a. Gail E. Kaiser and David Garlan, ''Melding Software Systems from Reusable Building Blocks,'' *IEEE Software (USA)* **4**(4)IEEE, (July 1987).

Katoh86a. Hideki Katoh, Hiroyuki Yoshida, and Masakatsu Sugimoto, *Logic-Based Retrieval and Reuse of Software Modules,* IEEE (1986).

Onuegbe87a. Emmanual O. Onuegbe, ''Software Classification as an Aid to Reuse: Initial Use as Part of a Rapid Prototyping System,'' pp. 521-52 in *Proceedings of the Twentieth Annual Hawaii International Conference on System Sciences,* (1987).

Pietri87a. F. Pietri, P. P. Puncello, P. Torrigiani, G. Casale, M. Degli Innocenti, G. Farrari, G. Pacini, and F. Turini, ''ASPIS: A Knowledged-Based Environment for Software Development,'' pp. 375-391 in *ESPRIT '87 Achievements and Impact, Procs. 4th Annual ESPRIT Conf., Part 1, Brussels, Sept 28-29 1987,* Commission of the European Communities (1987).

Ramamoorthy86a. C. V. Ramamoorthy, Vijay Garg, and Atul Prakash, ''Programming in the Large,'' *IEEE Transactions on Software Engineering* **SE-12**(7) pp. 769-783 IEEE, (July 1986).

Rich83a. Charles Rich and Richard C. Waters, ''Formalizing Reusable Software Components,'' in *Proc. ITT Workshop on Reusability in Programming, 7-9 Sept 1983, Newport, RI,* ITT Programming (1983).

Tracz87a. Will Tracz, ''RECIPE: A Reusable Software Paradigm,'' pp. 546-555 in *Proceedings of the Twentieth Annual Hawaii International Conference on System Sciences,* (1987).

Wegner83a. P. Wegner, ''Varieties of Reusability,'' pp. 30-44 in *Proc. ITT Workshop on Reusability in Programming, 7-9 Sept 1983, Newport, RI,* (1983).

Woodfield86a. Scott N. Woodfield, David W. Embley, Gary L. Stokes, and Khang Zhang, *Assumptions and Issues of Software Reuseability,* IEEE (1986).