This article appeared in 1st Australian Requirements Engineering Workshop, ARE'96, Dept of Information Systems, Monash University, Caulfield Campus, Australia, September 1996, pp 2.1-17.

Reusing Informal Requirements: Review of Methods and Techniques

Jacob L. Cybulski
Department of Information Systems
The University of Melbourne
Parkville, Vic 3052
Phone: +613 9344 9244, Fax: +613 9349 4596
Email: j.cybulski@dis.unimelb.edu.au

Abstract

Reuse of software products resulting from the early phases of the development life-cycle is claimed to have a tremendous impact on the reduction of cost and enhanced productivity in software development [22]. As requirements engineering sets off the entire development process, consequently the reuse of its products will offer most significant benefits. Unfortunately, software requirements documents are still produced in the form and media which are inappropriate for subsequent computer representation and processing, not to mention their reuse. Besides, the lack of formality in the communication of software requirements is not only demanded by non-technical personnel, but in fact can also have some positive effects on requirements validation and traceability. And so the process of reusing software requirements will have to deal with the analysis of informal texts into their formal representation, their storage and organisation, and the synthesis of the new requirements documents of reusable informal components. To this end, the paper reviews numerous methods and techniques, from a variety of different computing disciplines, which could assist this process and enhance the reusability of requirements engineering products.

1. Introduction

As in many other approaches to requirements engineering, we admit the need to capture software requirements in the form and media most appropriate for the problem domain and convenient to the user community. We also advocate that such collections of mostly informal requirements should be analysed, refined and ultimately formalised, so that other conventional methods were possible in the ensuing development process. Where we may differ to many other researchers is our conviction that the best way of improving the cycle of requirements engineering is via reuse of previously processed requirements documents, whether formal or informal. And to this end, the most effective method is to provide requirements engineers with the reuse tools which could assist them in the analysis, refinement and formalisation of informal requirements while, at the same time, utilising the skills, knowledge and experience of the people intimately involved in the process [20].

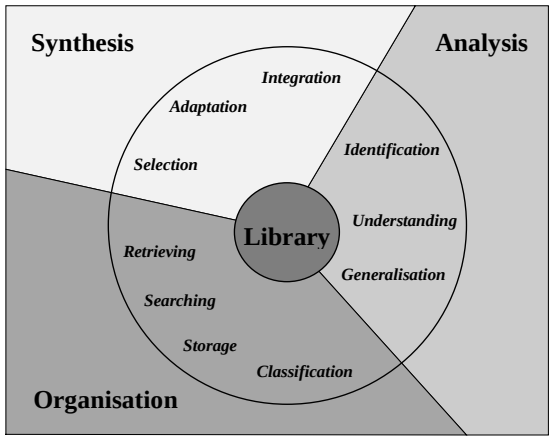

Figure 1 - The reuse process

In this paper, we view the task of software reuse as comprising three stages of artefact processing (cf. Figure 1) :- the analysis of artefacts and their characteristics, their organisation into a reuse library and the

Reuse Phase	**Approach**	**Techniques**	
identification	code rejuvenation	static text analysis	*
		text restructuring	
		text generation	
	domain analysis	customised artefact editors	*
		domain vocabulary	*
		semantic representation	*
		classification of domain concepts	*
		capture of strategies and tactics	
		capture of reasoning knowledge	
		system commonalties	
understanding	media properties	document surface analysis	
		lexical analysis	*
	artefact usage assessment	statistical usage properties	
	artefact classification	taxonomies	*
		facets	*
	adding text descriptors	text indexing	*
		information retrieval	*
		natural language processing	*
	structure representation	data dictionaries	
	knowledge representation	knowledge representation	*
		knowledge acquisition	
		inference engine	
		intelligent assistants	
generalisation	reasoning	analogical reasoning	
		induction and deduction	

Table 1 - Requirements analysis techniques[1]

subsequent synthesis of the new artefacts out of the library components [21]. The reuse process is equally applicable to code, designs and specifications, however, in the following sections we will investigate how the three aspects of software reuse-in-general apply to the effective reuse of informal software requirements, the most cumbersome life-cycle products.

2. Artefact Analysis

The analysis phase of software reuse starts with the process of *identifying* reusable components in existing software products and in the description of a problem domain. Once a collection of artefacts and their properties had been identified, they need to be *understood*, i.e. represented in a formalism suitable to reflect their function and semantics. Before artefacts could be stored in a reuse library, first however, they may have to be *generalised* to widen the scope of their future application. A number of techniques used in the process of identification, understanding and generalisation of requirements documents are listed in Table 1, the techniques are further discussed in the following sections.

2.1 Artefact Identification

The majority of practical effort in the identification of reusable software components can only be described as ad-hoc foraging of existing software repositories, in search of useful programs, modules, functions or data. The more systematic methods of identifying software artefacts emerge from the work on software rejuvenation [72, 453ff] and domain analysis [4]. Both of these methods can be used to facilitate reuse of requirements specifications.

---

[1] Note that asterisks indicate those of the features adopted by the author in his development of an editor for the acquisition and analysis of informal requirements [24].

Software Rejuvenation. The first of the two approaches aims specifically at improving software documentation, structure, design and quality in general, so that its maintenance could be performed with greater ease. There are four methods of achieving this goal :-

- *Redocumentation* - static analysis of source code which results in additional information that could help maintainers to better understand and reference code.
- *Restructuring* - transformation of ill-structured code into better structured programs. Several commercial and research products are available to assist computer-aided code restructuring.
- *Reverse engineering* - recreating design, specification and other early software products from code that preceded it. Many forward engineering CASE tools provide facilities for the automatic derivation of low-level specifications and designs from programs written in C/C++, Ada, Fortran and Cobol.
- *Reengineering* - a combination of reverse and forward engineering to alter existing programs, improve their quality or add new functionality, in absence of written specifications and designs.

Both reverse engineering and re-engineering rely on program understanding which is often facilitated by static analysis of source code, modularisation, restructuring of the flow of control, conversion of restructured code into a design language, adaptation of designs to remove undesirable code and design features, and generation of new code [9]. Code rejuvenation techniques, and in particular reverse engineering and redocumentation, commonly leads to the production of data dictionaries, process specifications, module-calling hierarchies, pseudocode, entity-relationship charts, data and control flow diagrams, structure charts, module and variable tables, cross reference, data interface tables, test paths, etc [72, p 455, 458].

Rejuvenated software can be significantly improved in its reusability characteristics by isolating its replaceable features, making products self-contained, parametrising, enriching, abstraction and specialisation, testing and validating, formally verifying, assessing quality, and restructuring [1]. At the same time, technology limits the scope of practical software rejuvenation, in particular that of reengineering and reverse engineering. While restructuring usually operates within a single layer of software abstraction, e.g. design-to-design or code-to-code, redocumentation, reengineering and reverse engineering rely heavily on the reconstruction of abstractions higher than those encapsulated in the available artefacts, e.g. code-to-design or design-to-specifications. Although, the three approaches have certain successes in commercial applications [63 - REFINE/Cobol, 70 - Cobol/SRE], the current state of technology still seems to be quite immature for the more complex tasks, like derivation of high-level software abstractions from code - the amount of development information lost in the process of consecutive software refinements from the original specifications down to code is far too great. To recover lost information, some researchers use rich, and frequently less structured and less formal, knowledge of the problem domain which can then be combined with the structures extracted from legacy code to arrive at a high-level, abstract specification of analysed programs [10, 27, 87]. This leads us to the integration of techniques drawn from software rejuvenation and the second approach to information identification in artefact analysis, i.e. domain analysis.

Domain Analysis. Domain analysis is the process of identifying and organising knowledge about some class of problems -- the problem domain -- to support their description and the solution of these problems [5]. Typical approaches to domain analysis include :-

- *Characterisation of the problem domain.* One of the most important aspects of domain analysis is to construct a domain model being a systematic and comprehensive description of all reusable resources useful in solving domain problems. Typically, such a

model specifies an extensive vocabulary of terms describing objects, their attributes and relationships characteristic to the problem domain. The vocabulary can then be used to represent library components in terms of their design properties [57] and to classify domain objects to improve their access and utilisation in the reuse tasks [76].

- *Modelling operational knowledge.* The domain model would not be complete without knowledge of development processes which create, refine and alter domain artefacts. Such knowledge may be observed as strategies and tactics invoked during artefact development and whenever needed reapplied and reused in the construction of new software products [68, 69]. It may also be worth considering the representation and reasoning mechanisms as reusable resources themselves being an integral part of the component library [82].
- *Exploration of software systems.* Existing software systems developed to solve problems drawn from the shared application domain can also be the source of knowledge about the domain and its objects. It is a common practice to study software components to discover and exploit commonality of their features, representations and methods [51, 53]. Other types of development documents, such as manuals, system documentation or even interview transcripts may also be analysed with a view to identify domain objects, relationships and operations, their similarities and abstractions [88]. The total knowledge about software development can be organised into distinct domains defined as classes of similar systems, and all of the reusable resources characterising domains can be classified in terms of features related to design rationale, information on functionality, performance, development methodology, operational constraints, etc [8].

The new systems requirements can be specified formally almost entirely in terms of a domain model, its descriptors and operators. The informal software requirements, in turn, can be analysed for references to domain concepts and relationships, and subsequently they could be refined into a composition of reusable formal domain objects. With the aid of a domain model, it is also possible to reverse engineer reusable code components back into their specifications, which themselves may also have reuse potential. The tools typically employed in all of the above methods are knowledge-acquisition tools, entity-relationship and object-oriented modeling tools, sematic clustering, CASE and various text parsing tools [1].

2.2 Artefact Understanding

Once a requirement document or its components has been identified, its features and sub-components need to be characterised, its relationships with other software products described, and its applicable operations specified. All this information will subsequently have to be embedded in some kind of high-level, symbolic and abstract description. Ideally, such a description should be in a form available for further machine processing leading to improved requirements classification, search and retrieval, interfacing, decomposition and integration. In the past, with varying degrees of success, the following approaches were used to capture artefact characteristics.

Media Properties. Some of the approaches to software reuse rely on the properties of media used to encode the software products rather than on the syntax or semantics of the artefacts themselves. For example, requirements can be classified using textual, lexical or syntactic properties instead of the semantic properties of encoded information.

Usage. Requirements documents can be characterised by their usage, access or modification profiles. Some systems attempt to determine requirements utility from the frequency of its reuse by developers. Others look at developer's behaviour during the search for applicable artefacts, the acceptance and rejection of the proposed candidates. Then again, some approaches focus on the artefact adoption and adaptation patterns to determine useful relationships between artefact collections.

Classification. Requirements artefacts may be characterised through the taxonomic or faceted classification of their properties and components.

Description. Because all work products, except code, are documents intended for humans, text reuse is becoming increasingly important [77]. This includes textual description of requirements in other media forms as well. Artefact descriptions are frequently expressed in natural language, such as English. Such descriptions may be processed with a number of standard text manipulation methods, e.g. text indexing, information retrieval or natural language processing.

Structure. Structured requirements documents, e.g. those produced by diagramming tools or formal specification languages may be described in terms of the objects and attributes they directly encode. Such descriptions can them be stored in a system data dictionary for further processing and reuse.

Knowledge Representation. While formal notations and systems are adequate for formalisation of artefact features which are well defined, complete and cohesive, they are not flexible enough to capture the richness of all types of information processed in the course of software development, e.g. information contained in the informal communication between clients and requirements engineers, information used by software designers in the process of problem solving, or the information used as the basis of fault finding by software maintainers. To cope with the new challenges of artefact complexity, researchers go beyond the realms of structured descriptions towards the experimental knowledge representation of software.

In the past, such knowledge bases of artefact representations have been used in :-

- *modeling domain knowledge* which could subsequently be used to guide reuse of software components in later design tasks [78, 84];
- *software requirements acquisition* into a knowledge base of domain knowledge, design methods, decision making processes, and knowledge required for checking model consistency [91].
- *requirements modeling* through the use of knowledge representation of, and formal reasoning about, domain agents, their goals, functional and non-functional requirements, abstract domains and time [11, 45]; or
- *assisting design tasks* through the representation and use of reusable rules for design specialisation and refinements, domain-oriented data object descriptions, domain attributes for data objects and design schemas, reusable templates and code [58, 78].

2.3 Artefact Generalisation

Once an artefact has been identified and understood, whether in the process of domain analysis, artefact construction or reverse engineering, it may have to be generalised to increase its scope of reusability. Such generalisation may be achieved by relaxing or removing design constraints imposed on artefacts, by integrating implied or redundant design concepts with the main problem solution, by dealing with more abstract data structures, or by substituting the selected component of an artefact with a more general one. Very little work on tool assisted artefact generalisation has been done to date. Theoretically, systems assisting developers in reasoning by analogy [39, 61, 62] or cases [55] and program proving techniques [48] seem to be appropriate for this task but solutions in this area are still largely unexplored.

Reuse Phase	Approach	Techniques	
storage & retrieval	relational representation	tables	
	hierarchical representation	taxonomies	
	software repository	CASE data dictionary	*
	CMS	configuration management	
	knowledge base	facts and rules	*
class. & search	query language	boolean logic	
		relational algebra	
		phonetic variants	
		menus	*
		restricted natural language	*
		component descriptors	*
		graphic browsers	
	information retrieval	keyword search	*
		signature matching	
		faceted and taxonomic classification	*
		thesaurus	
		lexical affinity indexing	
	hypertext	explicit links	*
		forms and templates	*
		hierarchies	*
		pattern matching	*
		technology books	

Table 2 - Requirements organisation techniques[2]

3. Artefact Organisation

Software requirements, to be useful in the reuse process, need to be organised into a systematic repository of components. Before *storage*, artefacts may have to be *classified* into a taxonomy of concepts based on their conceptual properties expressed in an appropriate notation, e.g. propositional networks, frames or rules. Alternatively they may have to be *indexed* with respect to some controlled or uncontrolled vocabulary or be classified in terms of knowledge representation elements. The organisation process could then use tools allowing *searching* and *retrieval* of artefacts. Such tools may be capable of context verification, assessing semantic closeness of artefacts, using thesauri to determine term similarity, applying boolean or vector space information retrieval methods, utilising either query systems, hypertext or hierarchical browsers, employing sematic and citation searching tools, etc. [1].

The techniques useful in storage, retrieval, classification and search of requirements documents are listed in Table 2 and they are further discussed in the following sections.

3.1 Artefact Storage and Retrieval

The methods employed in the storage and retrieval of software artefacts go far beyond the issues of record storage and database management. Such methods will vary as the nature of software components varies due to their formalisation, meaning, structure, form or media (Cf. Table 3).

- *Tables and taxonomies.* The simplest, albeit most established, techniques put to use in the storage and retrieval of software are those used with binary and source code, e.g. link-loader tables [79] and taxonomies of code modules and classes [43].

---

2 Note that asterisks indicate those of the features adopted by the author in his development of an editor for the acquisition and analysis of informal requirements [24].

Enterprise:	**Application design:**	**Validations and verification:**
o Organisational structure	o Methodology rules	o Test plan
o Business area analyses	o Graphical representations	o Test data cases
o Business functions	o System diagrams	o Regression test scripts
o Business rules	o Naming standards	o Test results
o Process models (scenarios)	o Referential integrity rules	o Statistical analyses
o Information architecture	o Data structures	o Software quality metrics
	o Process definition	
Project management:	o Class definition	**System documentation:**
o Project plans	o Menu trees	o Requirements documents
o Work breakdown structure	o Performance criteria	o External/internal designs
o Estimates	o Timing constraints	o User manuals
o Schedules	o Screen definitions	
o Resource loading	o Report definitions	**Construction:**
o Problem reports	o Logic definitions	o Source code
o Change requests	o Behavioural logic	o Object code
o Status reports	o Algorithms	o System build instructions
o Audit information	o Transformation rules	o Binary images
		o Configuration dependencies
		o Change information

Table 3 - Types of repository information [74, p 752]

- *Data dictionaries.* The more sophisticated techniques are those used in the representation of diagrams and charts commonly used in software designs and specifications, e.g. CASE data dictionaries [46].
- *Text databases.* Efficient storage and retrieval of unstructured and informal documents, such as requirements statements, problem reports or change requests, is still in its infancy, and often relies on various techniques used in text processing [80].
- *Integrated repositories.* Repositories of software artefacts collected over the entire development life-cycle fulfil functions additional to normal data management, i.e. interoperability of tools accessing such a repository, ensuring non-redundancy and consistency of information contained in artefacts of differing levels of abstraction, exchange of data between different development tools, and synchronisation of tools via the repository [12, 86].
- *Knowledge-bases.* The sophistication of data stored in reuse libraries and the complexity of services demanded of such a library lead to suggestions of employing specialised software information systems [35, 36] or knowledge-based systems [59, 78, 82].

3.2 Artefact Classification and Searching

Artefact classification and the subsequent search is tightly coupled with the underlying repository storage and retrieval mechanisms. We will consider here the main variations in query systems, information retrieval techniques and hypertext.

Query Languages. A typical library of reusable software artefacts, whether consisting of requirements, specifications or code, is organised around an existing database management system. In such a library, requirements can be accessed via one of the standard query languages based on relational algebra, as in SQL or QBE. Other systems, e.g. CATALOG [33], allows handling queries in boolean logic, returning not only the exact matches but also some phonetic variants. On occasions, library enquiries issued menus or restricted natural language are mapped into a regular query system, e.g. in RSL [14].

There are different methods of identifying artefact searchable attributes, some based on attribute-value pairs, others on enumerative taxonomies, faceted classification or simple keywords. The conducted experiments show that search methods seems to be of a far lesser

importance, than the choice of searchable attributes, their organisation and use [34]. This phenomenon is best illustrated by the architecture of the most powerful artefact retrieval systems which employ rich semantics of component descriptors. Such descriptors are capable of capturing artefact structural and behavioural properties, their dependencies and relationships. Effective retrieval of reusable components can then be achieved by formulating a suitable query involving various interrelated component descriptors [15].

Some researchers argue that simple attribute-value or descriptor systems cannot deal adequately with certain important aspects of artefact representation, e.g. their semantic proliferation, multiple views, and the need for intelligent indexing. In LaSSIE, a knowledge based system of reusable software components, a sophisticated frame-based representation is used to describe actions (e.g. external and internal actions, and stimuli), objects (e.g. communication device, resource, hardware and software), doers (e.g. users, processes, groups and processors), and states (e.g. line, resource, data and network states). In such a complex artefact repository, queries and browsing actions are issued in concert via LaSSIE's graphical and natural language user interfaces. Their resolution is, in turn, assisted by the sophistication of semantic matching and reasoning [29].

Information Retrieval. Software artefact retrieval based on simple query-answering is severely limited in its usefulness in dealing with elaborate artefact types, such as requirements. Chen [17] observes the following deficiencies of the query systems :-

- keyword-based systems are inadequate because they are based on lexical and syntactical properties of components rather than their semantics;
- there is a lack of standard formalism or schemata for the description of complex reusable components;
- more complex component, such as requirements specifications or designs, can only be adequately described in a suitable modeling language;
- components should be abstracted based on the linguistic properties of their name or their description;
- the retrieval system should not abandon its search when the direct query fails; instead it should offer some closely related components which do not exactly match the query.

Consequently, Chen suggests a model of retrieval based on the semantic representation of reusable components and the signatures matching of the goal specification and the artefacts stored in the database. The method proposed closely resembles those commonly used in databases and information retrieval [80]. Many other information retrieval methods permit treating software artefacts as plain unstructured text or as a vector of artefact properties, both eminently suitable to manipulation of complex requirements artefacts.

Prieto-Diaz [75, 76] adopts a faceted classification scheme (similar to that used in library cataloguing). His retrieval process is subsequently based on the metric assessing conceptual closeness of attributes in each of the facets, supported with the use of a thesaurus.

The REUSE system [7] provides a customisable, keyword-based hierarchical menu system being a front-end to the information retrieval system. Such menus and keywords provide a consistent classification of reposited software artefacts and are used to construct information retrieval queries. The REUSE system also maintains a thesaurus to reduce terminology differences within the user community.

In GURU [60], artefact attributes are automatically extracted from natural language artefact descriptions with the use of an indexing scheme based on the notions of lexical affinities and quantity of information. Subsequently, a browsing hierarchy is generated using an attribute clustering technique. Thanks to the free-text indexing scheme, GURU can also

accept free-style natural language queries treated simply as another artefact specification needed to be matched and compared against the rest of the artefact database.

Hypertext. An attractive form of organising and accessing the texts of software requirements documents is hypertext - a web of interconnected text units which can be browsed and accessed via navigational links represented in the body of text as selectable text buttons. The main benefit of hypertext-based retrieval is the visual and explicit nature of inter and intra document relationships. Such relationships can either be defined directly by the analysts or indirectly by running an appropriate text indexing and linking utility program. Several existing software engineering research projects offer hypertext as a viable facility for efficient access to software documents.

In DIF and ISHYS [36, 38], all software documents, to include code but also specifications, are defined as forms filled with information, organised into a tree-structure of templates and their instances. DIF documents can subsequently be retrieved by navigating the hierarchy of documents, following the semantic links or querying the selected keywords.

HyperCASE [23] provides a visual, integrated and customisable software engineering environment, supporting software developers in the areas of project management, system analysis, design and coding via a suite of loosely-coupled tools supporting both textual and diagrammatic presentation techniques. Tool integration is achieved by combining a hypertext-based user interface with a common knowledge-based document repository in a manner which facilitates integrity and completeness checking, component and design re-use, configuration management, etc.

KIOSK, the system for structuring of, and navigating between, UNIX text files, uses a similar method of accessing its text components. It further extends its hyper-navigation using a simple form of pattern matching [19].

While the previous projects aim at providing a flexible access to individual project components, EUROWARE aims at accessing reusable information over wide area networks (e.g. World Wide Web on the Internet), thus, providing a vehicle for sharing reusable artefacts across projects, companies or even countries [47].

Technology books [6], are another form of consolidating development knowledge, into a collection of structured and navigable documents. Such organisation of development information permits document classification according to their types (domain entities, project entities, work products, resources, statements, and analyses) and setting relationships between information nodes to form semantic taxonomies (history, taxonomy, aggregation, use, justification, intersection, and ownership). Access to this information is given via a number of semantic (issue, definition, assumption, constraints, position, design, unresolved and result) and syntactic (author, heading, equation, enumeration) tags.

4. Artefact Synthesis

The construction or synthesis of software products of reusable components starts with finding and retrieving several software artefacts matching the new software requirement. Subsequently the most suitable artefacts are then *selected* from amongst candidates and then *adapted* to suit the new application. Finally a range of selected software components are *integrated* into a completely new software product. The techniques for the selection, adaptation and integration of requirements artefacts are listed in Table 4 and are discussed below in more detail.

Reuse Phase	Approach	Techniques	
selection	copying	cut-change-paste	*
integration		copy libraries	
adaptation	embedding	document inclusion	
		sub-documents	
	referencing	modularisation	
		parametrisation	
		customisation	
		hypertext links	*
		dynamic linking & embedding	
	expansion	macros	
		glossaries	
		fields	
		print-merge	
	specialisation	objects & classes	
		templates	*
		outlines	*
		forms	
		document masters	*
	transformation	cross-abstration generators	
		history reuse	*

Table 4 - Requirements synthesis techniques[3]

Within the development-with-reuse, new artefacts can be constructed by incorporating reusable items by means of variant selection, instantiation, provision of data by generation and forms management, template completion by prompters and macro expansion, modification, and integration by linkers, smart editors, environments with integration paradigms, etc [1]. A selection of these methods can be adapted (and in some cases has been adapted) for the construction of requirements specification documents, sometimes with the aid of tools as simple as wordprocessor or as complex as knowledge-based systems.

Copying. Anecdotal accounts of software development practices appear to indicate that artefact copying is the most predominant approach to software reuse. Such observations are not surprising when the studies of cognitive behaviour of programmers reveal component copying to be a vivid exhibition of mental laziness displayed by the majority of software developers, whether experienced or novices [83]. To capitalise on this seemingly natural behaviour, or perhaps to restrain it, some programming languages, such as COBOL, offer source code copy-libraries (the precursors to the more general macros), which prevent a totally ad hoc approach to copy-reuse and regulate access, integration and modifications applied to copied code. While the syntax and semantic of modern programming languages promotes other forms of reuse, e.g. based on the taxonomies of abstract data types or classes, the informal organisation of early software artefacts, such as requirements statements, would still encourage the most primitive and irregular reuse methods, i.e. find, cut, change and paste of the selected text.

Embedding. Inclusion of previously developed artefacts into newly constructed documents seems to be a small advance over the cut-change-and-paste approach to text-reuse. For years, it has been used as a vehicle for embedding of shared text in new documents, e.g. in C header files or Microsoft Word sub-documents. Artefact embedding is a particularly useful form of sharing and reuse for documents with ill-defined or informal syntax, structure and unpredictable type of embedded components, such as in informal requirements texts.

---

3 Note that asterisks indicate those of the features adopted by the author in his development of an editor for the acquisition and analysis of informal requirements [24].

Referencing. By far the most common technique of bringing together independent programming resources is via parametrised referencing of routines and modules, also known as procedure calls, inter-module communication or message passing. Virtually all modern programming languages feature some sort of program modularisation and parametrisation which allow not only integration of reusable software components but also to some extent their customisation via parameters. Similar facilities also exist in some formal specification languages and software development environments, e.g. parametrised traits and function signatures in Larch or OBJ [42, 48] or parametrised conceptual frameworks in Resolve [13, 30, 71]. Needless to say such facilities are also available during the construction of less formal requirements documents which may utilise generic hypertext-style text referencing [19, 28, 32, 37, 38] or documents dynamic linking and data exchange, as in Microsoft OLE [49], Apple publish-and-subscribe and a more recent OpenDoc [3], or other document integration technologies similar to that in CORBA [67].

Other forms of artefact referencing and interconnection are not immediately applicable to software requirements documents as they rely on the active nature of interfaced components, e.g. UNIX pipes, Common Lisp and Scheme streams, software buses [40], or library interconnection languages such as LIL [41].

Expansion. A form of text parametrisation which could be readily applied to the integration of both formal and informal texts are macros. The specific value of macros to the reuse effort is best illustrated by the IBM BB/LX system [56]. Macros are commonly used across the entire spectrum of programming languages and pre-processors, e.g. assemblers, Fortran in-line statements, C defines, UNIX shell aliases, etc. Larch trait inclusion has many macro-like features. Its semantics however goes beyond simple text substitution. Although wordprocessing documents frequently feature facilities such as glossaries, fields, or massive print-merge text substitution, their application to the reuse of encoded requirements information is very limited.

Specialisation. A more promising suite of reuse techniques is associated with artefacts reuse by their specialisation from a more generic software component. This approach is rapidly gaining the attention of software reusers object-orientated programming, design and analysis [64]. As object classes represent programming concepts with the elements of their specifications, hence, the practical reuse efforts were applied to the entire spectrum of software life-cycle, e.g. programming systems such as MELD [52], RESOFT [18], and REBOOT [66], HOOD design method [25], derivation of programs from domain knowledge [50], reuse of semantic data model [73], etc. What is less apparent, however, is the fact that the approach to artefact instantiation as adopted by object-oriented programming languages, such as Ada, Smalltalk, C++ or Eiffel, is more suitable in the reuse of executable artefacts (such as programs) or highly regular artefacts (such as database objects), but it does not apply so effectively to declarative and irregular (such as program specifications) or informal artefacts (such as software requirements).

Techniques which may be more suitable for free text specialisation include text filling and structuring with templates as in equation editors [65], program editors [2, 85, 89] or flowchart editors [81]. Similar techniques are being applied not only to the structural properties of instatiated texts but may also be used with the program semantics as well [54, 90]. Standard wordprocessing utilities such as document outlines, forms and document masters or templates may serve as an excellent vehicle for the encoding of reusable components of requirements texts.

Transformations. The transformational methodology may utilise two different approaches to reuse: the first which relies on the code generation from a specification written in a high-level language; and the second which allows reuse of previously observed developmental steps in the new software construction [31]. The former of the approaches can be exemplified with the generation of code from abstract program specifications in GIST,

SETL, V or ELI [16] and the latter with the work by Goldberg and De Antonellis on the reuse of development history and other types of temporal information contained in the reuse components [26, 44]. Although research into very high level programming languages (VHLL) resulted in a number of successful experimental tools and techniques, to date there are no practical methods of converting high-level real-life requirements specifications into executable software. Nor does there exist tools which could span software refinement leading from informal requirements down to code. Where the two methods could be useful, however, but had not been tested extensively, is to use both of the transformational methods in relation to a high-level informal requirements document and its formal specification. At such a narrowly confined development phase the refinement steps could be easily tracked, analysed and repeated as the need arises.

5. Summary and Conclusions

This paper reviewed, contrasted and summarised the methods and techniques useful in the processing of informal software requirements texts. It classified all of the approaches into three categories corresponding to the three steps in the reuse cycle, i.e. artefact analysis (cf. Table 1), organisation of artefact libraries (cf. Table 2) and synthesis of the new artefacts (cf. Table 4).

The main techniques considered useful in the process of requirements reuse were adopted from other computing fields, e.g. reverse engineering, domain analysis, information retrieval, data and knowledge-based systems, hypertext and natural language processing, object-oriented systems, compiler and editor construction, etc. Several examples of reuse in the small and reuse in the large illustrated the most successful applications of the selected techniques, e.g. the use of domain vocabulary to classify software artefacts, knowledge-bases to acquire and represent semantics of software components, natural language processing to identify reuse concepts in plain-text documents, information retrieval and hypertext to access and navigate reuse libraries, forms and templates to create new software artefacts, using artefact history to reapply previously performed development steps, etc.

Such successful methods and techniques, as collected and described in this paper, helped the author in the design and prototyping of RARE IDIOM/SoDA, a software environment for the creation, elaboration, analysis, refinement and reuse of informal software requirements documents [24]. The system provides requirements engineers with a template-based editor, which allows them to enter the text of informal requirements, analyse it with a natural language parser into a knowledge-base of reuse descriptors, refine it into a requirements specification, and freely hyper-navigate between various requirements documents to enhance the documents access, clarity of contained concepts and to improve tracability of the subsequently developed software artefacts to their original statement of need. In the future, RARE IDIOM/SoDA will be further extended and improved to better unify with other life-cycle support tools, such as diagram editors and data dictionaries, thus forming an integrated CASE environment - HyperCASE [23].

6. References

[1] Agresti, W.W. and F.E. McGarry, "The Minnowbrook Workshop on Software Reuse: A summary report". In *Software Reuse: Emerging Technology*, W. Tracz, Editor. Computer Society Press: Washington, D.C. 1988, p. 33-40.

[2] Ambler, A.L. and M.M. Burnett, "Influence of visual technology on the evolution of language environments". In *Visual Programming Environments: Paradigms and Systems*, E.P. Glinert, Editor. IEEE Computer Society Press: Los Alamitos, CA. 1990, p. 19-32.

[3] Apple, *OpenDoc Programmer's Guide*: Addison-Wesley Publishing Co., 1996.

[4] Arango, G., "Domain analysis methods". In *Software Reusability*, W. Schafer, R. Prieto-Diaz, and M. Matsumoto, Editors. Ellis Horwood: London, Great Britain. 1994, p. 17-49.

[5] Arango, G. and R. Prieto-Diaz, "Part1: Introduction and Overview, Domain Analysis and Research Directions". In *Domain Analysis and Software Systems Modeling*, R. Prieto-Diaz and G. Arango, Editors. IEEE Computer Society Press: Los Alamitos, California. 1991, p. 9-32.

[6] Arango, G., E. Schoen, and R. Pettengill, "Design as evolution and reuse". In *Advances in Software Reuse: Selected Papers from the Second International Workshop on Software Reusability*, Lucca, Italy: IEEE Computer Society Press. , 1993, p. 9-18.

[7] Arnold, S.P. and S.L. Stepoway, "The reuse system: Cataloging and retrieval of reusable software". In *Software Reuse: Emerging Technology*, W. Tracz, Editor. Computer Society Press: Washington, D.C. 1988, p. 138-141.

[8] Bailin, S., *KAPTUR: A Tool for the Preservation and Use of Engineering Legacy*. Technical Report 20852, CTA Inc.: Rockville, MD. 1991.

[9] Bennett, K., *et al.*, "Software maintenance". In *Software Engineer's Reference Book*, J.A. McDermid, Editor. Butterworth-Heinemann Ltd: Oxford, U.K. 1991, p. 20/1-18.

[10] Biggerstaff, T.J., B.G. Mitbander, and D.E. Webster, "Program understanding and the concept assignment problem", *Communications of the ACM* **37**(5), 1994: p. 72-82.

[11] Borgida, A., S. Greenspan, and J. Mylopoulos, "Knowledge representation as the basis for requirements specifications", *IEEE Computer*, April 1985: p. 82-90.

[12] Brown, A.W., P.H. Feiler, and K.C. Wallnau, "Past and future models of CASE integration". In *Fifth International Workshop on Computer-Aided Software Engineering*, : IEEE. , 1992, p. 36-45.

[13] Buci, P., *et al.*, "Part III: Implementing components in RESOLVE", *ACM SIGSOFT Software Engineering Notes* **19**(4), 1994: p. 40-51.

[14] Burton, B.A., *et al.*, "The reusable software library", *IEEE Software* **4**(4), 1987: p. 25-33.

[15] Castano, S. and V. De Antonellis, "The F3 Reuse Environment for Requirements Engineering", *ACM SIGSOFT Software Engineering Notes* **19**(3), 1994: p. 62-65.

[16] Cheatham, T.E., Jr., "Reusability through program transformations". In *Software Reusability: Concepts and Models*, T.J. Biggerstaff and A.J. Perlis, Editors. ACM Addison Wesley Publishing Company: New York, New York. 1989, p. 321-336.

[17] Chen, P.S., R. Henicker, and M. Jarke, "On the retrieval of reusable components". In *Advances in Software Reuse: Selected Papers from the Second International Workshop on Software Reusability*, Lucca, Italy: IEEE Computer Society Press. , 1993, p. 99-108.

[18] Cheng, J., "Reusability-based software development environment", *ACM SIGSOFT Software Engineering Notes* **19**(2), 1994: p. 57-62.

[19] Creech, M.L., D.F. Freeze, and M.L. Griss, "Using hypertext in selecting reusable software components". In *Hypertext'91*, San Antonio, Texas: Acm. , 1991, p. 25-38.

[20] Cybulski, J.L., *The formal and the informal in requirements engineering*. Research Report 96/5, The University of Melbourne, Department of Information Systems: Melbourne. 1996.

[21] Cybulski, J.L., "Reuse in the eye of its beholder: cognitive factors in software reuse". In *OzCHI'96*, Hamilton, New Zealand: IEEE Press. November, 1996.

[22] Cybulski, J.L., *Sharing and reuse in the development of information systems*. Research Report 96/4, The University of Melbourne, Department of Information Systems: Melbourne. 1996.

[23] Cybulski, J.L. and K. Reed, "A hypertext-based software engineering environment", *IEEE Software* **9**(2), March 1992: p. 62-68.

[24] Cybulski, J.L. and K. Reed, *The Use of Templates and Restricted English in Structuring and Analysis of Informal Requirements Specifications*. Research Report TR024, Amdahl Australian Intelligent Tools Programme, La Trobe University: Bundoora. 1993.

[25] D'Alessandro, M., P.L. Iachini, and A. Martelli, "The generic reusable component: an approach to reuse hierarchical OO designs". In *Advances in Software Reuse: Selected Papers from the Second International Workshop on Software Reusability*, Lucca, Italy: IEEE Computer Society Press. , 1993, p. 39-46.

[26] De Antonellis, V., S. Castano, and L. Vandoni, "Building reusable components through project evolution analysis", *Information Systems* **19**(3), 1994: p. 259-274.

[27] DeBaud, J.-M., B. Moopen, and S. Rugaber, "Domain analysis and reverse engineering". In *International Conference on Software Maintenance, .* , 1994, p. 326-335.

[28] Delisle, N. and M. Shwartz, "Neptune: A hypertext system for CAD applications". In *ACM SIGMOD Int. Conf. on Management of Data*, Washington, D.C. , 1986, p. 132-143.

[29] Devanbu, P., *et al.*, "LaSSIE: A knowledge-based software information System", *Cacm* **34**(5), 1991: p. 34-49.

[30] Edwards, S.H., *et al.*, "Part II: Specifying components in RESOLVE", *ACM SIGSOFT Software Engineering Notes* **19**(4), 1994: p. 29-39.

[31] Feather, M.S., "Reuse in the context of a transformation-based methodology". In *Software Reusability: Concepts and Models*, T.J. Biggerstaff and A.J. Perlis, Editors. ACM Addison Wesley Publishing Company: New York, New York. 1989, p. 337-359.

[32] Fletton, N.T., "A hypertext approach to browsing and documenting software". In *Hypertext: State of the Art*, R. McAleese and C. Green, Editors. Intellect Ltd: Oxford, England. 1990, p. 193-204.

[33] Frakes, W.B. and B.A. Nejmeh, "An information system for software reuse". In *Tutorial on Software Reuse: Emerging Technology*, W. Tracz, Editor. IEEE Computer Society Press: Washington, D.C. 1988, p. 142-151.

[34] Frakes, W.B. and T.P. Pole, "An empirical study of representation methods for reusable software components", *IEEE Transactions on Software Engineering* **20**(8), 1994: p. 617-630.

[35] Fugini, M.G., O. Niestrasz, and B. Pernici, "Application development through reuse: the Ithaca tools environment", *ACM SIGOIS Bulletin* **13**(2), 1992: p. 38-47.

[36] Garg, P.K. and W. Scacchi, "On designing intelligent hypertext systems for information management in software engineering". In *Hypertext '87*, North Carolina, USA: The Association for Computing Machinery. , 1987, p. 409-432.

[37] Garg, P.K. and W. Scacchi, "ISHYS: Designing an intelligent software hypertext system", *IEEE Expert* **4**(3), 1989: p. 52-63.

[38] Garg, P.K. and W. Scacchi, "Hypertext system to manage software life-cycle documents", *IEEE Software* **7**(3), 1990: p. 90-98.

[39] Gentner, D., "The mechanisms of analogical learning". In *Similarity and Analogical Reasoning*, S. Vosniadou and A. Ortony, Editors. Cambridge University Press: Cambridge, U.K. 1989.

[40] Gisi, M.A. and C. Sacchi, "A positive experience with software reuse supported by a software bus framework". In *Advances in Software Reuse: Selected Papers from the Second International Workshop on Software Reusability*, Lucca, Italy: IEEE Computer Society Press. , 1993, p. 196-203.

[41] Goguen, J.A., "Reusing and interconnecting software components", *IEEE Computer*, February 1986: p. 17-28.

[42] Goguen, J.A., "Principles of parametrised programming". In *Software Reusability: Concepts and Models*, T.J. Biggerstaff and A.J. Perlis, Editors. ACM Addison Wesley Publishing Company: New York, New York. 1989, p. 159-225.

[43] Goldberg, A., "The influence of an object-oriented language on the programming environment". In *Interactive Programming Environments*, D. Barstow, H. Shrobe, and E. Sandewall, Editors. McGraw-Hill. 1984, p. 141-174.

[44] Goldberg, A., "Reusing software development". In *Fourth ACM SIGSOFT Symposium on Software Development Environments*, Irvine, California: ACM Press. , 1990, p. 107-119.

[45] Greenspan, S., J. Mylopoulos, and A. Borgida, "Capturing more world knowledge in the requirements specification". In *6th International Conference on Software Engineering*, : The Institute of Electrical and Electronics Engineers. , 1982, p. 231-240.

[46] Group, I.E.P., *Introducing PCTE+* IEPG TA-13, GIE Emeraude. 1989.

[47] Gutierrez, G.S., "EUROWARE, una solucion para fomentar la reusabilidad en redes de area extendida". In *1st Conf. on Software Engineering*, Puerto de la Cruz. June, 1995.

[48] Guttag, J.V. and J.J. Horning, *Larch: Languages and Tools for Formal Specifications*. New York: Springer-Verlag, 1993.

[49] Harris, L., *Teach Yourself OLE Programming in 21 Days*. Indianapolis: SAMS, 1994.

[50] Iscoe, N., "Domain-specific reuse: an object-oriented and knowledge-based approach". In *Software Reuse: Emerging Technology*, W. Tracz, Editor. IEEE Computer Society Press. 1988, p. 299-308.

[51] Jaworski, A., *et al.*, *A Domain Analysis Process*. Technical Report Domain-Analysis-90001-N V 01.00.03, Software Productivity Consortium: Herndon. 1990.

[52] Kaiser, G.E. and D. Garlan, "Melding software systems from reusable building blocks", *IEEE Software* **4**(4), 1987: p. 17-24.

[53] Kang, K.C., *et al.*, *A Reuse-Based Software Development Methodology* CMU/SEI-92-SR-4, Software Engineering Institute. 1992.

[54] Katz, S., C.A. Richter, and K.-S. The, "PARIS: A system for reusing partially interpreted schemas". In *Software Reusability: Concepts and Models*, T.J. Biggerstaff and A.J. Perlis, Editors. ACM Addison Wesley Publishing Company: New York, NY 1989, p. 257-274.

[55] Kolodner, J.L., "Improving human decision making through case-based decision aiding", *AI Magazine* **12**(2), 1991: p. 52-68.

[56] Lenz, M., H.A. Schmid, and P.F. Wolf, "Software reuse through building blocks", *IEEE Software*, July 1987: p. 34-42.

[57] Lubars, M.D., "Domain analysis and domain engineering in IDeA". In *Domain Analysis and Software Systems Modeling*, R. Prieto-Diaz and G. Arango, Editors. IEEE Computer Society Press: Los Alamitos, California. 1991, p. 163-178.

[58] Lubars, M.D. and M.T. Harandi, "Addressing software reuse through knowledge-based design". In *Software Reusability: Concepts and Models*, T.J. Biggerstaff and A.J. Perlis, Editors. ACM Addison Wesley Publishing Company: New York, New York. 1989, p. 345-377.

[59] Lubars, M.D. and N. Iscoe, "Frameworks versus libraries: a dichotomy of reuse strategies". In *WISR6 6th Annual Workshop on Software Reuse*, Owego, N.Y. Nov., 1993.

[60] Maarek, Y.S., D.M. Berry, and G.E. Kaiser, "An information retrieval approach for automatically constructing software libraries", *IEEE Transactions on Software Engineering* **17**(8), 1991: p. 800-813.

[61] Maiden, N. and A. Sutcliffe, "Analogical matching for specification reuse". In *6th Annual Knowledge-Based Software Engineering Conference*, Syracuse, New York, USA: IEEE Computer Society Press. , 1991, p. 108-116.

[62] Maiden, N.A. and A.G. Sutcliffe, "Exploiting reusable specifications through analogy", *Communications of the ACM* **35**(4), 1992: p. 55-64.

[63] Markosian, L., *et al.*, "Using an enabling technology to reengineer legacy systems", *Communications of the ACM* **37**(5), 1994: p. 58-70.

[64] Meyer, B., "Reusability: the case for object-oriented design", *IEEE Software*, March 1987: p. 50-64.

[65] Microsoft, *Microsoft Word User's Guide: Word Processing Program for the Macintosh, Version 5.0*. Usa: Microsoft Corporation, 1991.

[66] Morel, J.-M. and J. Faget, "The REBOOT environment". In *Advances in Software Reuse: Selected Papers from the Second International Workshop on Software Reusability*, Lucca, Italy: IEEE Computer Society Press. , 1993, p. 80-88.

[67] Mowbray, T.J. and R. Zahavi, *The Essential CORBA: System Integration Using Distributed Objects*. New York, NY: Wiley, 1995.

[68] Neighbors, J.M., "The Draco approach to constructing software from reusable components", *IEEE Trans. on Software Eng.* **SE**(10), 1984: p. 564-574.

[69] Neighbors, J.M., "Draco: A method for engineering reusable software systems". In *Software Reusability: Concepts and Models*, T.J. Biggerstaff and A.J. Perlis, Editors. ACM Addison Wesley Publishing Company: New York, New York. 1989, p. 295-320.

[70] Ning, J.Q., A. Engberts, and W. Kozaczynski, "Legacy code understanding", *Communications of the ACM* **37**(5), 1994: p. 50-57.

[71] Ogden, W.F., *et al.*, "Part I: The RESOLVE framework and discipline - a research synopsis", *ACM SIGSOFT Software Engineering Notes* **19**(4), 1994: p. 23-28.

[72] Pfleeger, S.L., *Software Engineering: The Production of Quality Software*. Second ed. New York: Macmillan Pub. Co., 1991.

[73] Poulin, J., "Integrated support for software reuse in computer-aided software engineering (CASE)", *ACM SIGSOFT Software Engineering Notes* **18**(4), 1993: p. 75-82.

[74] Pressman, R.S., *Software Engineering: A Practitioner's Approach*. 3 ed. New York, N.Y.: McGraw-Hill, Inc., 1992.

[75] Prieto-Diaz, R., "Classification of reusable modules". In *Software Reusability: Concepts and Models*, T.J. Biggerstaff and A.J. Perlis, Editors. Addison-Wesley Pub. Co.: New York, NY. 1989, p. 99-123.

[76] Prieto-Diaz, R., "Implementing faceted classification for software reuse", *Cacm* **34**(5), 1991: p. 88-97.

[77] Prieto-Diaz, R., "Status report: Software reusability", *IEEE Software*, May 1993: p. 61-66.

[78] Puncello, P.P., *et al.*, "ASPIS: a knowledge-based CASE environment", *IEEE Software*, March 1988: p. 58-65.

[79] Reed, K., *Factors Influencing the Design, Implementation, Performance and Use of Linking Loaders*. Msc Thesis , Monash University: Clayton. 1983.

[80] Salton, G., *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*. Readings, Massachusetts: Addison-Wesley Pub. Co., 1989.

[81] Siemens, *XperCASE(C)*. . Siemens AG Austria: Vienna, Austria. 1993.

[82] Simos, M.A., "The growing of organon: a hybrid knowledge-based technology and methodology for software reuse". In *Domain Analysis and Software Systems Modeling*, R. Prieto-Diaz and G. Arango, Editors. IEEE Computer Society Press: Los Alamitos, California. 1991, p. 204-221.

[83] Sutcliffe, A. and N. Maiden, "Specification reusability: why tutorial support is necessary". In *Software Engineering 90*, Brighton, U.K.: Cambridge University Press. , 1990, p. 489-509.

[84] Tamai, T., "Applying the knowledge engineering approach to software development". In *Japanese Perspectives in Software Engineering*, Y. Matsumoto and Y. Ohno, Editors. Addison-Wesley Publishing Company: Singapore. 1989, p. 207-227.

[85] Teitelbaum, T. and T. Reps, "The Cornell Program Synthesizer: A syntax-directed programming environment", *Communications of ACM* **24**(9), 1981: p. 563-573.

[86] Thomas, I. and B.A. Nejmeh, "Definition of tool integration for environments", *IEEE Software* **9**(2), March 1992: p. 29-35.

[87] Thomson, R., K.E. Huff, and J.W. Gish, "Maximising reuse during reengineering". In *3rd International Conference on Software Reuse: Advances in Software Reusability*, Rio de Janeiro, Brazil: IEEE Computer Society Press. November, 1994, p. 16-23.

[88] Vitaletti, W. and E. Guerrieri, "Domain analysis within the ISEC Rapid Centre". In *8th Annual National Conference on Ada Technology*, . March, 1990.

[89] Volpano, D.M. and R.B. Kieburtz, "The template approach to software reuse". In *Software Reusability: Concepts and Models*, T.J. Biggerstaff and A.J. Perlis, Editors. ACM Addison Wesley Publishing Company: New York, New York. 1989, p. 247-256.

[90] Waters, R.C. and Y.M. Tan, "Toward a design apprentice: Supporting reuse and evolution in software design", *ACM Software Engineering Notes* **16**(2), 1991: p. 33-44.

[91] Zeroual, K., "KBRAS: a knowledge-based requirements acquisition system". In *6th Annual Knowledge-Based Software Engineering Conference*, Syracuse, New York, USA: IEEE Computer Society Press. , 1991, p. 38-47.