A Compendium of Formal Techniques for Software Maintenance

Jonathan P. Bowen * Peter T. Breuer † Kevin C. Lano ‡

Oxford University Computing Laboratory
Programming Research Group
11 Keble Road, Oxford OX1 3QD, UK.

June 1993

Abstract

Software maintenance is an important area in practical software engineering that has been largely overlooked by many theoretical computer scientists. This paper gives an overview of some formal techniques that have been developed recently to aid the software maintenance process, and in particular reverse engineering and re-engineering. In the future, it is suggested that specifications rather than programs should be maintained. The work described provides a mathematical basis to a large collaborative project that has been investigating many other aspects of software maintenance as well.

Keywords: Software maintenance, formal methods, reverse engineering, decompilation, re-engineering, formal specification, Z notation, object-oriented techniques.

To appear in the *BCS/IEE Software Engineering Journal.*

*Oxford University Computing Laboratory, Programming Research Group, 11 Keble Road, Oxford OX1 3QD, UK. Email: <Jonathan.Bowen@comlab.ox.ac.uk>.

†Escuela Técnica Superior de Ingenieros de Telecomunicación, Universidad Politéchica de Madrid, Edificio B, Ciudad Universitaria, E–28040 Madrid, Spain. Email: <ptb@dit.upm.es>.

‡Lloyd's Register of Shipping, 29 Wellesley Road, Croydon CR0 2AJ, UK. Email: <tcskcl@aie.lreg.co.uk>.

Contents

1 Introduction

The problem of software maintenance has been recognized as one of the most serious limitations on the application of computer technology [Fos90]. However, for many years it has been the ugly duckling of the computer science research community. Until relatively recently, little serious research has been undertaken in this area, with some notable exceptions (e.g., [Leh80]). For example, an article for a special section—there were not enough good submissions for a whole issue, as originally intended—in the *IEEE Transactions on Software Engineering* as recently as 1987 [Sch87] surveyed the number of articles that had appeared in the past few years, and found none in the previous 15 months.

The amount of time spent by industry in undertaking software maintenance is difficult to assess because companies are often secretive, and perhaps embarrassed, by such statistics; however, it is generally acknowledged to be up to 80% of effort involved with software, and rising [LPW88], although there are doubters of this view [Fos92]. Indeed, some parts of the computing industry do not have a great incentive to improve the situation since the bulk of their profits are gleaned from maintenance, and customers have little choice but to comply with maintenance contracts. Moreover, in university computer science curricula, there is often little emphasis on software maintenance and it is not seen as an interesting field of study by lecturers and students alike. It is usually considered more a necessary evil that can be dealt with when the time comes; and preferably as late as possible!

The support available for software maintenance is often limited in industry. Typically only simple, non-specialist tools are available, and little or no training is provided to personnel. Companies tend to rely on on-the-job experience or bought-in expertise instead, since the cost of staff is high and computing staff are particularly mobile, their skills being in high demand. Some toolsets aimed specifically at making application code more maintainable are coming onto the commercial market, but these are very expensive—often tens of thousands of dollars per license—and are often unsatisfactory in many ways, seemingly providing no mechanism by which users can store their unfolding understanding of the application as it is gained.

However, the status of research in this area is gradually improving. A quarterly journal specifically aimed at this topic, the *Journal of Software Maintenance: Research and Practice* has now been established. The IEEE organize conferences specifically on software maintenance. A *European Special Interest Group in Software Maintenance* has established a newsletter particularly for practitioners, to cover recent developments (e.g., [Fos92]). A *Centre for Software Maintenance* exists at Durham University in the UK; as well as undertaking research, much of it collaborative, it also organizes annual workshops. Two large collaborative ESPRIT projects, the REDO and MACS projects [Esp90], have recently acted as a focus of software maintenance research in Europe. The former has concentrated on *reverse engineering*, on the principle that applications are usually unmaintainable in the form in which they are presented for maintenance, and work has to be done in order to rediscover the required documentation and design information. This paper presents some of the research results obtained on this project. Information on other important research in this area may be found in [35].

In the rest of this paper, various facets of our research in the REDO project are sketched. The focal activity of the project was reverse engineering; the associated activities of decompilation and re-engineering were also considered. An important consideration is how software maintenance may be conducted in the future; it is suggested that maintenance should be undertaken at the specification level rather than the program level, and the two levels should be kept in step.

2 Overview of Techniques

The REDO project was established in 1989 to investigate the maintenance, validation and documentation of software systems [Kat90], with the following primary objectives:

1. develop methods to facilitate the maintenance, restructuring, validation, and transportation between different environments, of large software systems;
2. develop a comprehensive set of prototype tools for these activities;
3. articulate and develop a theoretical framework which will influence both the structure of the toolkit during the project, and the software development community thereafter.

The project has involved collaboration between eleven European industrial and academic partners. The diverse organizations involved have provided a wide range of talents and experience to the project. A compendium of the results of the project has been produced as a book [vanZ92], and constitutes the project's response to objective 3 above.

The Programming Research Group at Oxford University has been particularly involved in providing a formal foundation for the work of the project [6]. As well as providing mathematical underpinning for the techniques proposed by the project, we have also produced some prototype tools for the software maintenance process. The work has covered the areas of reverse engineering: decompilation, redocumentation and re-engineering; validation: post-hoc verification and generation of correct code from specifications; maintenance: new languages and methods to support maintenance, including new code generation techniques using our own advanced compiler-compiler technologies.

Work within the REDO project identified methodologies and techniques which seem to offer significant improvements in the maintenance of present and future applications. These techniques are based on a combination of formal methods, particularly high-level specification languages such as Z [Spi92], and on object-oriented design. Figure 1 gives an overview of the place of formal methods in our approach to reverse engineering.

Ultimately, the use of formal methods allows a better understanding of the meaning of existing code. Our activities have been directed towards making code maintainable, and we believe that this involves providing comprehensible and useful descriptions of modules, functions and data; 'useful' in the sense that they form a basis for further formally based transformations and re-use, correction, improvement or adaptation. The activity of maintenance then becomes a process of deriving, adjusting, and implementing specifications. Our tools provide the means to extract a specification from a section of code, deducing its form automatically where possible and providing support otherwise, and then aid the refinement of these specifications back to code. They also (and primarily) provide a means of reshaping specifications.

An initial loading and parsing stage involves the representation of procedural languages in a repository of semantic information (although purely syntactic information may be discarded). Further abstraction into a generic intermediate language, UNIFORM, can then be performed. This language supports the restructuring of control flow, such as the elimination of PERFORM THROUGH and GOTO constructs. From this representation, and derived information, an outline object-oriented design in the Z^{++} specification language (an object-oriented extension of Z) [26], can be extracted, possibly deliberately introducing abstraction away from the algorithmic detail and the structure of data. This design is then elaborated by further information derived from the code and existing documentation, to include the specification of methods acting on the data encapsulated in the class, and invariants capturing domain information regarding this data.

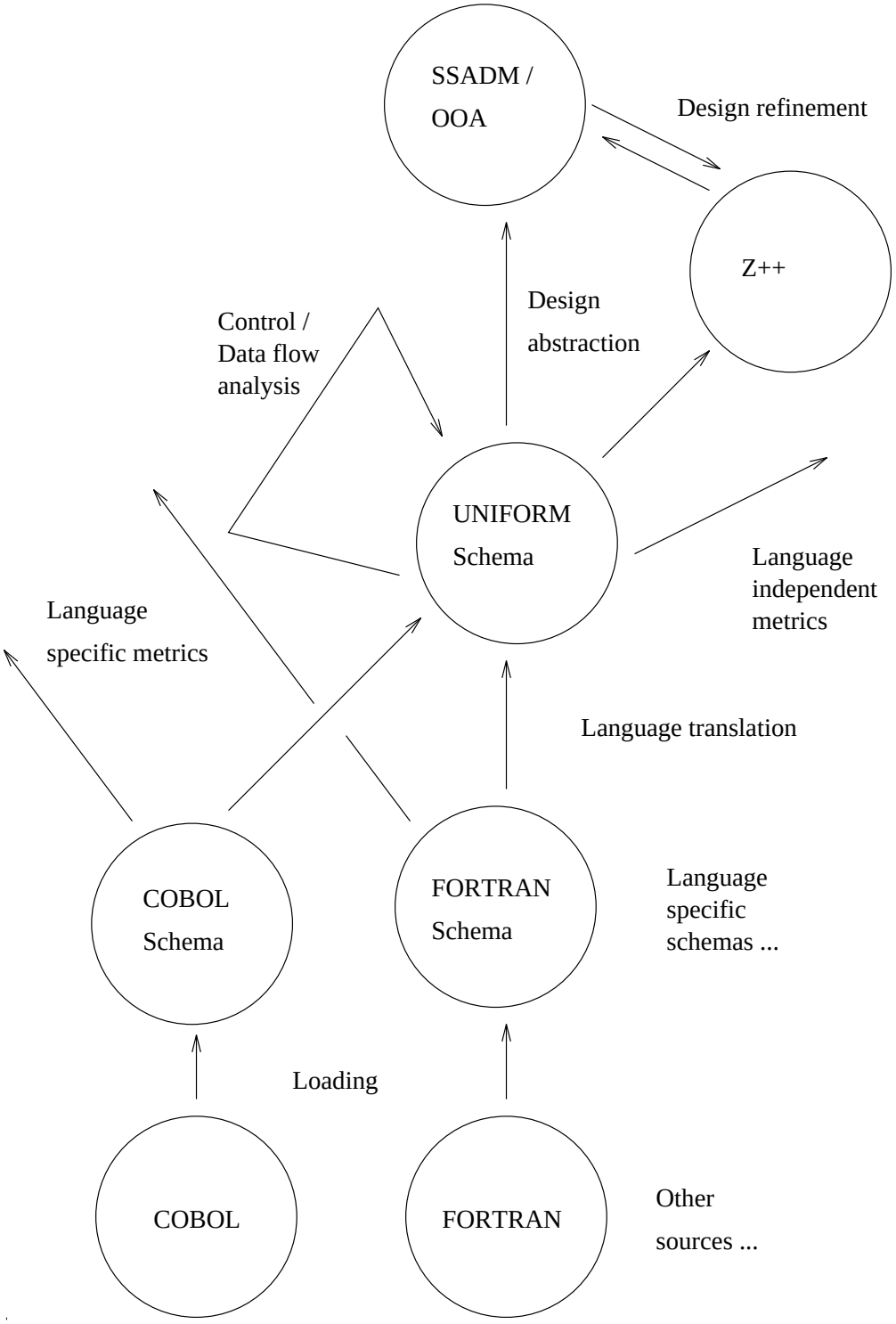


Figure 1: Process Model for Reverse Engineering

We support re-engineering at both the level of code and the level of specification. It is possible to abstract the functionality of a code fragment, to transform this specification into a simpler form, and then regenerate code from this specification, without detailed design analysis. This is feasible since close associations are maintained between the source code elements and their abstractions, so that only those parts of the source text which must change as a result of the changed specification are modified in the new code, with the remainder being retained. Alternatively, detailed design analysis and transformation could be performed, involving the incorporation of functional abstractions of code fragments into object classes as method specifications.

We believe that understanding of a complex program cannot come from the presentation of information without considerable human involvement [15]. Our methods and tools engage the maintenance engineer in an active dialogue with the program, involving, for example, the assertion of statements about the code and their subsequent validation or contradiction, and throughout the conversation, a specification level analogue of the code is being constructed that expresses what the code does and how it is organized, and how the specification relates to the parts of the code. The outcome is not merely a maintainable package, but understanding of that package by the maintenance engineer.

3 Defining a Semantics for UNIFORM

A key component of the REDO project has been the development of an intermediate language. Rather than handling COBOL, the major existing language for which a serious software maintenance problem exists, it was decided to use a simpler, more easily understandable notation with a well-understood semantics. COBOL itself has never been formalized in its entirety due to its complexity. However, by providing a translation to a formalized intermediate language, a formal semantics is then automatically obtained for the constructs that are translated. The semantics is not deterministic—many variants of COBOL will satisfy the semantics reflected by the translation—but reverse engineering tools can use the logic provided by the common abstraction.

In support of this methodology, a semantics for the intermediate language UNIFORM was developed, and provides a precise basis for verification and code transformation activities. A low-level denotational semantics was initially provided [20], and was the basis of a simple UNIFORM interpreter. An axiomatic semantics, more closely related to the predicate transformer semantics for programming languages which formed the basis for our verification method, was also produced [17, 23, 27]. Finally, a transformational semantics, by reduction of UNIFORM to Z^{++}, was produced [39, Chapter 5], [19], based on this axiomatic semantics, and formed the basis for the reverse engineering of COBOL via transformation to UNIFORM and specifications.

4 Advances in Parser Technology

In order to implement the first prototype version of the UNIFORM language, the experimental compiler-compiler tool *precc* was used [7, 10, 11]. This tool produces ANSI compliant C code for parsers with potentially infinite lookahead and backtracking and therefore is ideal for the rapid prototyping of language designs. Although the project as a whole decided to implement parsers for COBOL and UNIFORM using the standard *yacc* and *lex* utilities [JL78], the effort involved many man-years and almost the whole life-time of the project, whereas *precc* obtained a working parser for UNIFORM within a few man-days of effort by a single person. As it turned out, UNIFORM had to be redesigned to permit an implementation using *yacc*, in that extra keywords

had to be added which distinguished the beginnings and endings of different scopes from one another, and none of this was necessary for *precc*.

The *precc* compiler-compiler tool has latterly been enhanced, allowing the use of parameters to grammatical descriptions, and the integrated use of meta-parameters in generic constructions. Parsers for the whole of ANSI COBOL 74, and a demonstration parser for occam 2 [4] have now been constructed. These parsers are much faster than their equivalents in *yacc* would be (and *yacc* may not be able to handle all the features of the language conveniently and unaided); they are also highly maintainable because of the meta-grammatical constructs and modular construction. Grammar definition scripts for *precc* are conventionally split into different modules corresponding to the different parts of a language (simple statements, compound statements, declarations, processes, tokens, and so on) and compiled separately, which makes the turn-round time in maintenance activities very short, as only local changes in one or two modules may be required, or a module can be taken out of service while another is substituted in its place.

All these features make *precc* an attractive tool for producing maintainable front-ends to languages, and work on the utility continues. The tool is based on a well-understood higher-order theory and implements a precise specification, with which is associated an axiomatic semantics that supports formal proofs of what is or is not a properly phrased input to a parser. *Yacc* specifications are not easily amenable to such treatments because individual definitions in the script take their meaning from the context of the rest of the script.

5 The Z^{++} Specification Language

The concept of a 'process' in UNIFORM was recognized as being equivalent to the concept of an 'object' or instance of a 'class' under the object-oriented paradigm. This led to the development of the Z^{++} language as a unified abstract representation framework for expressing the functionality and design of applications, together with environment systems [12, 14, 29], and also led to tools for generating UNIFORM procedural code from Z^{++} specifications [16].

This language has been developed in concert with a precise definition of its axiomatic and model-based semantics. Only a brief description will be given here. An overview is given in [26], and [32, 30] provide details of the semantics. Methods of refinement and reasoning about Z^{++} specifications are given in [34], and application examples may be found in [12, 14, 29, 36], [37, Chapter 7].

The general layout of a Z^{++} specification is as follows:

Definitions of global types

Definitions of object classes

STATE

Variables used in every operation
Global invariants on these variables

Initial

$STATE'$
Initial values of variables

...
Definitions of operations

That is, a specification is a sequence of *paragraphs* of formal text (ideally interspersed, as in Z, with informal explanatory text paragraphs), which are either Z paragraphs in which class names are used as types, and class operations in method calls, or are class definitions. Full syntactic details of these definitions are given in [37, Chapter 2], [28].

6 Reverse Engineering

Early in the life of the project, the partners as a whole agreed to concentrate on *reverse engineering* as the fulcrum for their efforts. Exploratory research during the first year established that understanding an application was the major obstacle in rendering it maintainable.

The Oxford tools support reverse engineering as their primary aim. The principal process is that of transformation through a series of successively higher level languages, beginning with UNIFORM and culminating in a structured Z^{++} specification. Incoming COBOL code is first *cleaned*, and transformed to the equivalent UNIFORM code. During this process, certain details of the data representation may be lost, such as whether integers were stored as 16 or 32 bits, but all the essential functionality for understanding is retained.

The UNIFORM code is then abstracted to a first order functional language, in which details of the algorithms used are lost in favour of implicit representations of functionality. The functional language is then transformed to a representation in Z, during which more implementational details are lost, and used to populate the interiors of object designs which have been culled from the data-flow analysis at the UNIFORM and first order functional language levels (sections of UNIFORM code between I/O operations are candidates for analysis as individual units, and only the externally communicating variables in these units remain visible in the functional representation).

The process of transformation and code comprehension that we have developed can be summarized as follows (following the terminology of [12]):

Stage 1: Sanitize. The COBOL program is translated into the intermediate language UNIFORM, and redundant control structures are eliminated. `MOVE X TO Y` statements become assignments involving a format conversion, or *cast*: `Y := [FX→FY](X)` where `FX` is the declared format of `X`, `FY` of `Y`, and `[FX→FY]` is the casting function. The relationships between data are translated into logical *invariants* of the program.

Stage 2: Specify. Using data-flow diagrams, we group together associated variables to create outline objects. The code is split into *phases*, single input-output functions. We abstract the functions associated with these phases, `GO TO`s and other unstructured code constructs are eliminated, using user-guided transformation of functional abstractions or equivalent control-flow graphs.

Stage 3: Simplify. Simplifying transforms are applied to the abstracted functional descriptions, and to the derived object class hierarchy. The functions are incorporated into the object classes, as the definitions of the operations of these objects. A Z or Z^{++} description can be created from this object-oriented abstraction, together with other documentation, such as SSADM Entity Life Histories.

7 Decompilation

Decompilation from object code to source code may be considered part of the reverse engineering process, particularly if only the object code for a program is available. The project investigated two approaches to this problem. The first arose from research on the ESPRIT **ProCoS** project into proving compiling specifications correct [HHBP90]. The specifications turned out to be in the form of Horn clauses in general and thus could be coded conveniently and directly in a logic programming language like Prolog [1]. With some care, such programs may be formulated to act as either compilers, or decompilers, or even both, because of the relational rather than functional nature of logic programs [3].

An alternative, less direct but more efficient approach, resulted from attempting to cast the above technique in a functional programming setting, using a *decompiler-compiler* style of formulation. The techniques are compared in [5, 8, 9].

8 Validation and Verification

This area of research can be broken down into two main components:

- the *verification* of existing systems;
- the generation of *validated* code from formal specifications.

Prototype tools, which operated on the UNIFORM and Z languages, were developed for both of these areas. There is a close connection between reverse engineering of a piece of software, and validation of that software. Both can be seen as a process of obtaining information and understanding about the application. Validation means making sure that the application satisfies its requirements, which may be informal, or even the nebulous 'satisfies the customer'. Verification is defined to be a form of validation in which formal mathematical properties of the software are derived or proved, in particular, in which the functionality of the program is compared with a mathematical specification of its behaviour.

As a general observation, understanding of an application arises from the interaction of a user with the code through a toolset, and only a limited gain in understanding is to be expected from the presentation of information about the application alone, whether it be in the form of an abstract formalism or in the form of a diagram like a control-flow graph. The prototype tools attempt to engage the reverse engineer in an endeavour which increases comprehension of the logic of the application by involvement in its abstraction. In that sense, the logical formalisms which become attached to the application code serve as markers along a road to understanding, although they can be an end in themselves.

In general, reverse engineering efforts have not achieved the extraction of information at this level, being concerned more with overall design and graphical aids to code comprehension [Loo92]. However situations arise in practice in which detailed functionality is essential to the understanding and re-engineering of the code, and graphs do not help. This arose even in the case study of a library database system, described in Section 13, which was an almost entirely non-mathematical program, and these properties are clearly relevant in numerical processing or real-time domains. Our reverse engineering system can also be applied to tasks of verification. The documents [18, 24, 25, 33] contain more details of the tools.

9 Verification of Existing Systems

The problem addressed through this research is the support of formal verification activities on procedural (UNIFORM) source code, as defined above. One approach taken for the verification in practice of existing applications is based on the classical method of Floyd/Hoare assertions and Dijkstra's weakest preconditions [Hoa69], with additional elements for handling the concurrent programming constructs of UNIFORM. This was also the approach taken by the ESPRIT project ATES [CP90].

The innovative aspects of our work are in the use of heuristics to generate plausible loop-invariants and intermediate assertions [27], and the utilization of logic programming as a substrate, so that it becomes possible to manipulate incomplete assertions which become more fully established as work progresses. The tools are designed to support interaction with the user at those points where full automation is not possible, thus enabling the programmer's intuitions about procedural code to be used. A limited theorem-proving and algebraic simplification capacity is built into the system, using a decision procedure for Presburger arithmetic [Sho77].

In slightly more detail, Floyd/Hoare assertions are of the form

$$\{P\}\ cmd\ \{Q\}$$

asserting pre-condition P and post-condition Q for the code cmd in context. In terms of the weakest pre-condition semantics, the assertion is that P is strong enough to force $wp(cmd,\ Q)$, the weakest pre-condition which will ensure that the condition Q holds after cmd terminates. The conditions in braces become inserted into the code at appropriate points during the verification process, and the validative aspect of this work lies with the confirmation or refutation of more informal claims or guesses about the functionality. A limited amount of non-functional behaviour is also ascertainable in this way, since operations in the code may be assigned a time increment, or loading factor, and assertions about these parameters included in the conditions P and Q.

The method in practice involves splitting the code into subsections, asserting a desired post-condition Q for a given subsection cmd of a program, and using the tool to generate successive pre-conditions, working in reverse execution order through code statements until a pre-condition P for cmd with respect to Q is obtained. If cmd contains loops, then further proof conditions will be generated as a result of hypothesising loop invariants. The result is a logically consistent set of logical assertions, embedded in the code (these assertions are often sufficient as logic programming code in their own right, constituting an executable abstraction of the application in a higher level language). The logical programming substrate of the tool allows the post-condition and intermediate assertions to be initially unspecified, and refined automatically as the verification proceeds. This is an advantage in comparison with the similar tools from [CP90], for example.

Message passing constructs are handled as in Gypsy [Goo84], by regarding potential blockage at a `RECEIVE` or `SEND` communication statement as another form of process exit, and simultaneously deriving preconditions for required conditions to hold at these exit points, in addition to the exit at the logical end of a process. Thus even perpetual processes can be validated.

10 Generation of Code from Specifications

An area which has had a considerable amount of research and work devoted to it, and in which significant tools and systems already exist, is the automatic generation of procedural code from abstract (non-executable) specifications. Some systems, such as CIP [Bau85], use an interactive

transformational approach; others, like the B-tool [Abr92], use an automatic non-interactive approach. All are necessarily processes of *refinement* – from the abstract specification to the implementable pseudo-code. We have chosen the B-tool style of approach, as it reduces the work required by the user, while not substantially decreasing the power of the refinements.

A large example of application of our system, to a radar track-former, is given in [16]. At present the system would be useful for rapid prototyping, for executing proposed specifications in an exploratory way to ascertain their 'correctness' or credibility, rather than final development. The limits of validation for real systems have been pointed out in [Coh89]; we can never in fact completely formalize the design intention nor the actual executable code or device, so that any formal 'proof of correctness' is relative to a given set of simplifying assumptions or models. Therefore exploratory investigation is of definite value in certifying that the formal model does conform to the intentions, which may be unformalized and partial.

11 Generation of Entity Life Histories

From the object-oriented abstractions we can generate SSADM data-flow diagrams and entity life histories [AG90], where we equate entities to sets of variables. The user selects the variables to be examined, and the slice of the functional abstraction on these variables is calculated (so that some functions which do not affect the variables are discarded). The entity life history diagram for this variable group is then calculated from the *normalized* functional abstraction [33].

12 Design Extraction

Object-oriented designs are natural within the data-processing domain at least, where it is often the case that systems are implemented as layers of applications, with a higher level module of a system using a module from a lower level by means of its operations, but not having detailed access or knowledge of its internal functioning. Within the REDO project, object-oriented specifications were chosen as the most effective way of representing large application systems together with their environments, such as CICS or TOTAL. Abstract specifications of these systems, their available operations and the side-effects of these operations, are essential in fully capturing the functionality of an application. [21] gives a specification of parts of the CICS API in this style. The COBOL language itself has aspects, such as files, indexed arrays, subprograms, and the report writer, which can be directly represented as parameterized object classes [19]. The design which is abstracted from the code is built on top of these basic classes, and the reverse engineer should not normally need to examine the internal details of these.

Global invariants of the program are captured in the invariants of the abstracted object classes, the key functions of the code are captured as methods of the classes, and relationships of data-dependency and conceptual connection between variables can also be expressed in the invariants of these classes. Thus key aspects of an application can be expressed in a clear and comprehensible way in the formalism. Functional abstraction, while a useful tool in itself, needs to be used in an intelligent way, to avoid producing monolithic and incomprehensibly complex abstractions from raw source code. The application of abstraction only to code sections identified as housing meaningful operations at the global level is an effective way to break up the abstraction task.

13 Case Studies

During the REDO project one large data processing system was processed by our system. This is a suite of programs from the early 1970's [Peg91], which implemented a library database. It consisted of some 30,000 lines of code in total, although functional abstraction was restricted to smaller sections of code, with manual application in some places. The original code had the following flaws:

- Non-meaningful paragraph names;
- Unstructured control flow;
- No explicit invariant giving the logical relationship *Inv_Library* between the library database files;
- No parameterization of operations.

The improvements made in the restructured version were:

- Improved partitioning of paragraphs;
- Better names for paragraphs;
- Recognition of object-oriented design;
- Use of structured constructs and elimination of unstructured GO TO's;
- Parameterization of operations.

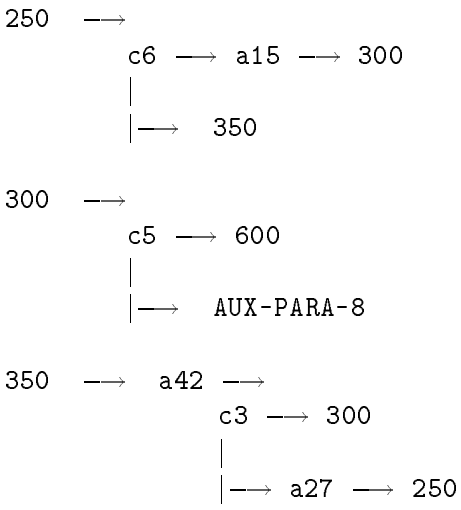
Classes associated with each of the three main database files `BOOK-FILE`, `COPY-FILE` and `TITLE-STACK-FILE` are recognized. Then, we merge these three classes into a single class (in theoretical terms, this is the co-product of classes), with the invariant *Inv_Library* expressing the connection between these files precisely given. This invariant was formalized from the documentation, and was verified using transformations on abstractions. Thus, we have the outline class:

```
CLASS Library_Database
OWNS
  BOOK-FILE : x13 ⇸ ddata × {'0', '1'};
  COPY-FILE : x13 × 0 .. 99 ⇸ cdata × {'0', '1'};
  TITLE-STACK-FILE : x50 × 0 .. 999 ⇸ x13 × {'0', '1'}
OPERATIONS
  DROP : CARD → ;
  CREATE : seq CARD →
INVARIANT
  Inv_Library
END CLASS
```

The type $x13$ is the set of sequences of 13 characters, and in the above, variables of this type hold ISBN numbers. The type $x50$ will hold book titles. The display formats have been retained in this case, due to their general significance for COBOL applications, although these can be abstracted if required.

In addition, there are two other classes, a *Card_File_Class* for the input stream and an *Error_Class* which handles the exception conditions.

As a small example of the abstraction process for program functionality, we can abstract the control flow graph / functional abstraction:

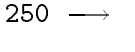


from the source code section:

```
       250.
           IF (ISBN-COPY NOT = ISBN-TITLE)
             DISPLAY BLANK-LINE UPON CONSOLE
             GO TO 300.
           GO TO 350.
      *
       300.
           IF (NCOPIES = 0)
                    GO TO 600.
           READ TITLE-STACK-FILE NEXT
                        AT END GO TO 400.
           GO TO 150.
      *
       350.
           MOVE ISBN-COPY TO ISBN-O.
           MOVE NUM-COPY TO COPY-O.
           MOVE LOCAT-COPY TO LOCAT-O.
           MOVE STAT-COPY TO STAT-O.
           ADD 1 TO NCOPIES.
           DISPLAY OUTPUT-LINE6 UPON CONSOLE.
           READ COPY-FILE NEXT
                    AT END GO TO 300.
           GO TO 250.
```

Labelled branches a_i stand for segments of straight line code, with no control flow branching.

The user can then rewrite the graph to the form:



```
c6 ⟶ a15 ⟶ 300
|
|⟶
      c3(a42()) ⟶ a42 ⟶ 300
      |
      |⟶ a42 ⟶ a27 ⟶ 250
```

Automatic regeneration of code can be performed, to produce the COBOL '85 code fragment:

```
250.
     MOVE 'F' TO VBL3.
     PERFORM UNTIL (VBL3 = 'T')
       IF (ISBN-COPY NOT = ISBN-TITLE)
        DISPLAY BLANK-LINE
        MOVE 'T' TO VBL3
       ELSE
        MOVE ISBN-COPY TO ISBN-O
        MOVE NUM-COPY TO COPY-O
        MOVE LOCAT-COPY TO LOCAT-O
        MOVE STAT-COPY TO STAT-O
        ADD 1 TO NCOPIES
        DISPLAY OUTPUT-LINE6
        READ COPY-FILE NEXT
          AT END
           MOVE 'T' TO VBL3
        END-READ
       END-IF
     END-PERFORM.
     PERFORM 300.
```

VBL3 can be replaced with a more meaningful name, such as EOF-COPY-FILE, and the code fragment incorporated into the *Library_Database* class (or, in fact, into the *COPY-FILEclass* which is an ancestor of this class). Inputs and outputs which are not local variables of the class become declared parameters of a method whose semantics is defined by the function corresponding to the abstraction of the paragraph:

$$SHOW\text{-}ALL\text{-}COPIES\text{-}OF\text{-}TITLE\ \ ISBN\text{-}TITLE\ \mid\ NCOPIES\ \texttt{==>}\ 250(ISBN\text{-}TITLE, NCOPIES)$$

The restructured version of the program was determined to be much easier to understand by the user organization, and tests devised for the software could not distinguish between the functionalities of the two versions.

The techniques used are supported by tools which are based on the use of the programmers intuition about program semantics, and on abstraction transformations based on this understanding (selected from a menu). They do not require great familiarity with mathematical proof or logic.

Work is continuing to adapt these techniques to other languages, particularly FORTRAN 77 and C.

14 A Rewrite System for Finite Process Descriptions

Work was also carried out in the area of re-engineering concurrent systems, defined in a language such as CSP [Hoa85]. A set of transformations and rewrite rules were developed and implemented [22], which removed all occurrences of the concurrent execution operator (||).

15 Maintenance Models

As part of the formal methods underpinnings for the REDO project, industrial collaboration with Computer Technologies Co. in Greece on the design and prototype implementation of a model of the software maintenance process was undertaken [40]. This model expresses the concepts embodied in the software maintenance department of a medium sized software house in terms of synchronized Predicate Nets, and was easily implemented in Prolog.

The prototype implementation of the model is able to predict the effectiveness of various management strategies that might be employed by the department 'against' a stream of incoming maintenance projects. Both the characteristics of the incoming stream and the characteristics of the model maintenance department can be customized to reflect the real situation fairly accurately, which makes the implementation a useful commercial tool, and it is being pursued as such by CTC. The prototype also allows the use of 'real' incoming jobs—suitably coded—and the ability to run various allocations of personnel and other resources in order to evaluate the best real-life options.

16 A Specification-based Approach to Maintenance

In this part of our research, we used the language Z^{++}, and a method based upon this language, to support the use of formal methods in software maintenance. The method is centered on the maintenance of the *specifications* and the *development record*, not upon source code or Structured Methodology documentation. It is proposed as a practical approach for software in the medium term future, allowing the mass of programming detail that makes the code maintenance problem so expensive to be ignored. Therefore changes and extensions to application systems can be made more rapidly. We describe the language and give details of the specification and refinement system, together with a description of the current state of the implementation of this system in [31].

The components of the method are:

- a standard specification style for systems;
- exact semantic correspondences between code and specification forms [27];
- a systematic process of implementing changes to systems by changing the specification and updating all documentation in line with this change.

In other words we intend to maintain the *development record*, the entire structure of refinement steps and documentation encompassing the derivation of code from specifications, as opposed to just maintaining the code. Because of the considerable investment already by some parts of industry in using and learning Z (e.g., [Phi90]), we felt such a formalism should be chosen, especially as it is widely regarded as one of the best or most usable formal specification languages, and much information and experience is increasingly available about the language [2].

The significant difference between our language and Z is in the ability to define object *classes*, templates for objects which encapsulate a state, invariant properties of that state, and operations, owned by the object, using this state. We also adopt a wide-spectrum approach; the use of code constructs of UNIFORM, such as `DO WHILE` and `RECEIVE / SEND`, which have a precise mathematical meaning [17].

Theoretical background for the approach, as applied with Z and the B Abstract Machine Notation, is described in [38]. An important consideration in the approach is the need to ensure that transformations upon a system as a result of a required maintenance change are refinements, and that the specification constructs preserve refinement.

17 Conclusion

This paper has outlined a number of techniques which could be used in the software maintenance process, and for the reverse engineering of existing programs in particular. Research at Oxford University has concentrated on the formal aspects of work undertaken on the collaborative REDO project. In addition, we have produced a number of prototype tools, mostly using Prolog [13]. These have been integrated around a common database with the other tools produced on the project. The overall toolset, or one similar to it, could eventually form the basis of a software maintainer's workbench. Industrial partners on the REDO project have integrated a considerable number of software maintenance tools onto a common platform (including those described in this paper). A significant number of publications have resulted directly from the work on the project and the most important of these are listed at the end of this paper.

Reverse engineering is essentially a human process requiring flair and skill [15]; tools can only aid the maintainer to gain an understanding of the code more quickly, not fully automate the generation of a specification from an existing program. Once a specification is obtained, a new program may be re-engineered from it using one of the more standard and well understood development techniques.

Having a specification of a program introduces the problem of maintaining the specification and keeping it in step with the program. This paper has presented a technique, based on an object-oriented version of the formal Z notation, known as Z^{++}, that could be used to aid the maintenance of programs. It is to be hoped that an increasing amount of software will be well specified in the future to allow such an approach to be adopted.

REDO tools and methods have been further developed by the industrial partners of the consortium, particularly the University of Limerick, via their campus company, Piercom, and by Lloyd's Register. Further projects, the DTI SREDM project, and the REM Eureka project, are extending the REDO methods.

Acknowledgements

We thank our colleagues on the ESPRIT II REDO project (no. 2487) for a stimulating and varied three years on the project. In particular, Howard Haughton of Lloyd's Register (London, UK), and Giorgos Papapanagiotakis, formally at CTC (Athens, Greece), have co-authored a number of reports and publications. The research was undertaken in the main whilst the authors were at the Oxford University Computing Laboratory. Peter Breuer and Kevin Lano were funded by the REDO project. Jonathan Bowen was funded by the UK Science and Engineering Research Council (SERC).

Bibliographies

Please note that copies of REDO project documents and PRG Technical Reports and Monographs are available from the Librarian at the Oxford University Computing Laboratory.

A Project Publications and Reports

[1] J.P. Bowen. From programs to object code using logic and logic programming. In R. Giegerich and S.L. Graham, editors, *Code Generation – Concepts, Tools, Techniques*, Workshops in Computing, pages 173–192. Springer-Verlag, 1992. Proc. International Workshop on Code Generation, Dagstuhl, Germany, 20–24 May 1991.

[2] J.P. Bowen. Select Z bibliography. In J.P. Bowen and J.E. Nicholls, editors, *Z User Workshop, London 1992*, Workshops in Computing, pages 309–341. Springer-Verlag, 1993.

[3] J.P. Bowen. From programs to object code and back again using logic programming: Compilation and decompilation. *Journal of Software Maintenance: Research and Practice*, to appear.

[4] J.P. Bowen and P.T. Breuer. Occam's razor: The cutting edge for parser technology. In *TOULOUSE 92: Fifth International Conference on Software Engineering and its Applications*, EC2, 269 rue de la Garenne, 92024 Nanterre Cedex, France, 7–11 December 1992.

[5] J.P. Bowen and P.T. Breuer. Decompilation. In H. van Zuylen, editor, *The REDO Compendium: Reverse Engineering for Software Maintenance*, chapter 9, pages 131–138. John Wiley, 1993.

[6] J.P. Bowen, P.T. Breuer, and K.C. Lano. The REDO project: Final report. Technical Report PRG-TR-23-91, Oxford University Computing Laboratory, 11 Keble Road, Oxford, UK, December 1991.

[7] P.T. Breuer. A PREttier Compiler Compiler: higher order programming in C. In *TOULOUSE 92: Fifth International Conference on Software Engineering and its Applications*, EC2, 269 rue de la Garenne, 92024 Nanterre Cedex, France, 7–11 December 1992.

[8] P.T. Breuer and J.P. Bowen. Decompilation *is* the efficient enumeration of types. In M. Billaud et al., editors, *Journées de Travail WSA'92 Analyse Statique*, volume BIGRE 81–82, pages 255–273, F-35042 Rennes cedex, France, 1992. IRISA-Campus de Beaulieu.

[9] P.T. Breuer and J.P. Bowen. Decompilation: the enumeration of types and grammars. Technical Report PRG-TR-11-92, Oxford University Computing Laboratory, 11 Keble Road, Oxford, UK, May 1992. Provisionally accepted by *ACM Transactions on Programming Languages and Systems (TOPLAS)*.

[10] P.T. Breuer and J.P. Bowen. A PREttier Compiler-Compiler: Generating higher order parsers in C. Technical Report PRG-TR-20-92, Oxford University Computing Laboratory, 11 Keble Road, Oxford, UK, November 1992. Submitted for publication.

[11] P.T. Breuer and J.P. Bowen. The PRECC compiler-compiler. In E. Davies and A. Findlay, editors, *Proc. UKUUG/SUKUG Joint New Year 1993 Conference*, pages 167–182, Owles Hall, Buntingford, Herts SG9 9PL, UK, 1993. UK Unix Users Group / Sun UK Users Group, UKUUG/SUKUG Secretariat.

[12] P.T. Breuer and K.C. Lano. Creating specifications from code: Reverse engineering techniques. *Journal of Software Maintenance: Research and Practice*, 3:145–162, 1991.

[13] P.T. Breuer and K.C. Lano. Using Prolog for reverse-engineering and validation. In *Logic Programming Summer School (LPSS '92), Zurich*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 1992.

[14] P.T. Breuer and K.C. Lano. Reverse engineering COBOL via formal methods. *Journal of Software Maintenance: Research and Practice*, 5, March 1993.

[15] P.T. Breuer, K.C. Lano, and J.P. Bowen. Understanding programs through formal methods. In H. van Zuylen, editor, *The REDO Compendium: Reverse Engineering for Software Maintenance*, chapter 15, pages 195–223. John Wiley, 1993.

[16] K.C. Lano. Validation through refinement and execution of specifications. REDO project document 2487-TN-PRG-1041, Oxford University, UK, August 1990.

[17] K.C. Lano. An axiomatic semantics of UNIFORM. REDO project document 2487-TN-PRG-1011, Oxford University, UK, December 1991.

[18] K.C. Lano. The design of the verification toolset. REDO project document 2487-TN-PRG-1068, Oxford University, UK, August 1991.

[19] K.C. Lano. Expressing the semantics of COBOL in Z. REDO project document 2487-TN-PRG-1055, Oxford University, UK, November 1991.

[20] K.C. Lano. An operational semantics for UNIFORM. REDO project document 2487-TN-PRG-1005, Oxford University, UK, December 1991.

[21] K.C. Lano. An outline specification of the CICS API. REDO project document 2487-TN-PRG-1025, Oxford University, UK, December 1991.

[22] K.C. Lano. A rewrite system for finite process descriptions. REDO project document 2487-TN-PRG-1082, Oxford University, UK, November 1991.

[23] K.C. Lano. Simple concurrent reasoning. REDO project document 2487-TN-PRG-1019, Oxford University, UK, December 1991.

[24] K.C. Lano. Test results for the verification tool set. REDO project document 2487-TN-PRG-1069, Oxford University, UK, August 1991.

[25] K.C. Lano. Verification using formal techniques. REDO project document 2487-TN-PRG-1066, Oxford University, UK, August 1991.

[26] K.C. Lano. Z^{++}, an object-orientated extension to Z. In J.E. Nicholls, editor, *Z User Workshop, Oxford 1990*, Workshops in Computing, pages 151–172. Springer-Verlag, 1991.

[27] K.C. Lano and P.T. Breuer. From programs to Z specifications. In J.E. Nicholls, editor, *Z User Workshop, Oxford 1989*, Workshops in Computing, pages 46–70. Springer-Verlag, 1990.

[28] K.C. Lano, P.T. Breuer, and H.P. Haughton. Using object-oriented extensions of Z for maintenance and reverse-engineering. Technical Report PRG-TR-22-91, Oxford University Computing Laboratory, 11 Keble Road, Oxford, UK, December 1991.

[29] K.C. Lano, P.T. Breuer, and H.P. Haughton. Reverse engineering COBOL via formal methods. In H. van Zuylen, editor, *The REDO Compendium: Reverse Engineering for Software Maintenance*, chapter 16, pages 225–248. John Wiley, 1993.

[30] K.C. Lano and H.P. Haughton. Axiomatic semantics for object-orientated specification languages. In *ZOOM Workshop, Oxford University*, 11 Keble Road, Oxford, UK, August 1991. Preprint.

[31] K.C. Lano and H.P. Haughton. A specification-based approach to maintenance. *Journal of Software Maintenance: Research and Practice*, 3:193–214, December 1991.

[32] K.C. Lano and H.P. Haughton. An algebraic semantics for the specification language Z^{++}. In *Proc. Algebraic Methodology and Software Technology Conference (AMAST '91)*. Springer-Verlag, 1992.

[33] K.C. Lano and H.P. Haughton. Extracting functionality and design from COBOL. In *Proc. CASE'92*. IEEE Press, 1992.

[34] K.C. Lano and H.P. Haughton. Reasoning and refinement in object-oriented specification languages. In O.L. Madsen, editor, *ECOOP '92: European Conference on Object-Oriented Programming*, volume 615 of *Lecture Notes in Computer Science*, pages 78–97. Springer-Verlag, 1992.

[35] K.C. Lano and H.P. Haughton. Software maintenance research and applications. In J. Leponiemi, editor, *Proc. NordDATA '92*. Pitky, 1992.

[36] K.C. Lano and H.P. Haughton. Integrating formal and structured methods in reverse engineering. In *Working Conference in Reverse Engineering*. IEEE Press, May 1993.

[37] K.C. Lano and H.P. Haughton. *Object Oriented Specification Case Studies*. Prentice Hall, 1993. To appear.

[38] K.C. Lano and H.P. Haughton. Reuse and adaptation of Z specifications. In J.P. Bowen and J.E. Nicholls, editors, *Z User Workshop, London 1992*, Workshops in Computing, pages 62–90. Springer-Verlag, 1993.

[39] K.C. Lano and H.P. Haughton. *Reverse Engineering and Software Maintenance: A Practical Approach*. International Series in Software Engineering. McGraw Hill, 1993. To appear.

[40] G. Papapanagiotakis and P.T. Breuer. A software maintenance management model based on queuing networks. *Journal of Software Maintenance: Research and Practice*, 4, 1992.

B Other References

[Abr92] Abrial J.-R., *Assigning Programs to Meaning*, Prentice Hall 1993, to appear.

[AG90] Ashworth C., Goodland M., *SSADM: A Practical Approach*, McGraw Hill International Series in Software Engineering, 1990.

[Bau85] Bauer F.L., *The Munich project CIP. Volume 1: the wide spectrum language CIP-L*, Springer-Verlag, Lecture Notes in Computer Science, **183**, 1985.

[Coh89] Cohn A., Comments on the formal proof of the VIPER microprocessor, *International Journal of Automated Reasoning*, **5** 2, 1989.

[CP90] Couturier P., Puccetti A., ATES: An integrated system for software development and validation, *ESPRIT '90 Conference Proceedings*, Kluwer Academic Publishers, 1990, 242–263.

[Esp90] ESPRIT Directorate General XIII, *Synopses of Information Processing Systems, ESPRIT II Projects and Exploratory Actions*, Volume 4 of a series of 8, Commission of the European Communities, September 1990.

[Fos90] Foster J., Those maintenance statistics, *Software Maintenance Workshop*, Centre for Software Maintenance, Durham University, September 1990.

[Fos92] Foster J., Survey report, *European Special Interest Group in Software Maintenance*, Newsletter Issue 3, June 1992, 5–7.

[Goo84] Good D.I., Mechanical proofs about computer programs, *Phil. Trans. Royal Society*, London, **A 312**, 1984, 389–409.

[JL78] Johnson S.C., Lesk M.E., Language development tools, *The Bell System Technical Journal*, **57** 6, part 2, July/August 1978, 2155–2175.

[Hoa69] Hoare C.A.R., An axiomatic approach to computer programming, *Communications of the ACM*, **12**, October 1969.

[Hoa85] Hoare C.A.R., *Communicating Sequential Processes*, Prentice Hall International Series in Computer Science, 1985.

[HHBP90] Hoare C.A.R., He Jifeng, Bowen J.P., Pandya P.K., An algebraic approach to verifiable compiling specification and prototyping of the ProCoS level 0 programming language, in Directorate-General of the Commission of the European Communities (eds), *ESPRIT '90 Conference Proceedings*, Kluwer Academic Publishers, 1990, 804–818.

[Kat90] Katsoulakos P.S., REDO, in Norman R.J., Ghent R.V. (eds), *CASE '90: Fourth International Workshop on Computer-Aided Software Engineering*, IEEE Computer Society Press, 1990.

[Leh80] Lehman M.M., Programs, life cycles, and laws of software evolution, *Proc. IEEE*, **68** 9, 1980, 1060–1076.

[LPW88] Leonard J., Pardoe J., Wade S., Software maintenance – Cinderella is still not getting to the Ball, in *Proc. Second IEE/BCS Conference: Software Engineering 88*, Conference Publication No. 290, 1988, 104–106.

[Loo92] Loosley, C., *CASE Tools and Repositories: The Challenge of Integration*, NordDATA '92, DataBase Associates, 1992.

[Peg91] Pegueroles J.M., *T5310 Final Information Package*, Centrisa, Barcelona, Spain, 1991.

[Phi90] Phillips M., CICS/ESA 3.1 experience, in Nicholls J. (ed), *Z User Workshop, Oxford 1989*, Springer-Verlag, Workshops in Computing, 1990, 179–185.

[Sch87] Schneidewind N.F., The state of software maintenance, *IEEE Transactions on Software Engineering*, **SE-13** 3, March 1987, 303–310.

[Sho77] Shostak R., The Sup-Inf method for proving Presburger formulae, *Journal of the ACM*, **24** 4, October 1977, 529–543.

[Spi92] Spivey J.M., *The Z Notation: A Reference Manual*, 2nd edition, Prentice Hall International Series in Computer Science, 1992.

[vanZ92] van Zuylen H. (ed), *The REDO Compendium: Reverse Engineering for Software Maintenance*, John Wiley, 1993.