Programming Research Group

THE REDO PROJECT: FINAL REPORT

Jonathan P. Bowen
Peter T. Breuer
Kevin C. Lano

PRG-TR-23-91

Oxford University Computing Laboratory
8–11 Keble Road, Oxford OX1 3QD
Abstract

This report gives an overview of the work performed by the Programming Research Group as part of the European collaborative ESPRIT II “REDO” project (no. 2487). This work covered the areas of reverse-engineering: redocumentation and re-engineering; validation: post-hoc verification and generation of correct code from specifications; maintenance: new languages and methods to support maintenance. Research in areas of concurrent programming and decompilation were also performed.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview of the Project</td>
<td>1</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>2</td>
</tr>
<tr>
<td>2 Defining a Semantics for UNIFORM</td>
<td>2</td>
</tr>
<tr>
<td>3 Validation and Verification</td>
<td>2</td>
</tr>
<tr>
<td>3.1 Verification of Existing Systems</td>
<td>3</td>
</tr>
<tr>
<td>3.2 Generation of Code from Specifications</td>
<td>3</td>
</tr>
<tr>
<td>3.2.1 Procedural refinement</td>
<td>4</td>
</tr>
<tr>
<td>3.2.2 Implementation results</td>
<td>5</td>
</tr>
<tr>
<td>4 Reverse-Engineering of Existing Applications</td>
<td>6</td>
</tr>
<tr>
<td>4.1 An Overview of the Method</td>
<td>7</td>
</tr>
<tr>
<td>4.2 The Reverse-Engineering Process</td>
<td>7</td>
</tr>
<tr>
<td>4.2.1 Stage 1: Translation from COBOL to UNIFORM</td>
<td>7</td>
</tr>
<tr>
<td>4.2.2 Stage 2: Higher level abstraction</td>
<td>8</td>
</tr>
<tr>
<td>4.2.3 Functional abstraction</td>
<td>8</td>
</tr>
<tr>
<td>4.2.4 Stage 3: Simplification of abstractions and design</td>
<td>8</td>
</tr>
<tr>
<td>4.2.5 Example: Reverse-engineering a scientific application program</td>
<td>9</td>
</tr>
<tr>
<td>4.2.6 Example cont’d: Reverse-engineering in the abstraction domain</td>
<td>12</td>
</tr>
<tr>
<td>4.2.7 Example cont’d: Data equivalences in reverse-engineering</td>
<td>12</td>
</tr>
<tr>
<td>4.3 Generation of Entity Life Histories</td>
<td>17</td>
</tr>
<tr>
<td>4.4 Design Extraction</td>
<td>17</td>
</tr>
<tr>
<td>4.4.1 Example: Design abstraction</td>
<td>19</td>
</tr>
<tr>
<td>4.5 A Rewrite System for Finite Process Descriptions</td>
<td>20</td>
</tr>
<tr>
<td>5 A Specification-based Approach to Maintenance</td>
<td>20</td>
</tr>
<tr>
<td>5.1 The Z(^{++}) Language</td>
<td>21</td>
</tr>
<tr>
<td>5.2 Metrics</td>
<td>22</td>
</tr>
<tr>
<td>5.3 Development and Refinement</td>
<td>22</td>
</tr>
<tr>
<td>5.4 Reusability</td>
<td>22</td>
</tr>
<tr>
<td>5.5 Summary</td>
<td>23</td>
</tr>
<tr>
<td>5.6 A Formal Model for Maintenance and Development</td>
<td>23</td>
</tr>
<tr>
<td>5.7 Formalising Class Transformations</td>
<td>25</td>
</tr>
<tr>
<td>5.8 Laws</td>
<td>26</td>
</tr>
<tr>
<td>6 Decompilation Techniques</td>
<td>27</td>
</tr>
<tr>
<td>6.1 Decompilation</td>
<td>27</td>
</tr>
<tr>
<td>6.2 Decompiler Compilers</td>
<td>29</td>
</tr>
<tr>
<td>6.3 Conclusions</td>
<td>31</td>
</tr>
<tr>
<td>6.3.1 Future problems</td>
<td>33</td>
</tr>
<tr>
<td>7 Conclusion</td>
<td>33</td>
</tr>
</tbody>
</table>

Acknowledgements i
## Bibliographies

A REDO Publications and Reports, Oxford

B Other References

## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Source code <strong>A</strong></td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>Restructured source code <strong>A</strong></td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>Stages in the production of a <em>literal</em> specification for source code <strong>A</strong></td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>Reasoning about the specification to achieve mathematical simplification</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>The relationship between stacks <strong>s</strong> and <strong>t</strong></td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>Mathematical reasoning on the equations describing code <strong>A</strong></td>
<td>18</td>
</tr>
<tr>
<td>7</td>
<td>Code for the final object</td>
<td>32</td>
</tr>
</tbody>
</table>
Overview of the Project

The collaborative ESPRIT II REDO project was established to investigate the maintenance, validation and documentation of software systems [Kat90]. The original primary objectives of the project were to:

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 the following eleven European partners:

- Lloyd’s Register of Shipping (London, UK) – the coordinators;
- Centrisa (Barcelona, Spain);
- CTC (Athens, Greece);
- Delft Hydraulics (The Netherlands);
- Durham University (UK);
- Electricité de France (Paris, France);
- Grumann Daten-Kommunikation GmbH (Bad Homburg, Germany);
- Marconi Command and Control Systems Ltd (Leicester, UK);
- Ingenieria y Tecnologia de Sistemas S.A. (Madrid, Spain);
- Oxford University (UK);
- University of Limerick (Ireland).

The project started in January 1989 and finishes at the end of December 1991 (at Oxford at least), although partners with funds available may continue work until April 1992. A handbook of the results of the project is in preparation [vanZ91].

At Oxford, Jonathan Bowen has acted as local project manager; Peter Breuer and Kevin Lano have been employed as full-time Research Officers on the project. The Programming Research Group has been particularly involved in providing a formal foundation for some of the work on the rest of the project. We have also produced some prototype tools for the software maintenance process. This report summarises the research undertaken at Oxford and provides a full bibliography of publications and reports produced as a result of this work.
1 Introduction

The problem of software maintenance has been recognised as one of the most serious limitations on the application of computer technology [Fos90]. 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, and on object-oriented design.

In this report we provide an overview of the techniques we have developed, and give references to examples of their use. The areas covered correspond approximately to the work-packages 1A (Theoretical development of “UNIFORM”, an intermediate language [SC90]), 2B (Reverse-engineering), 9 (Validation), and 11 (Concurrency and Maintenance of Future Applications) of the REDO project.

2 Defining a Semantics for UNIFORM

In order to provide a basis for verification and code transformation activities, a precise semantics for the intermediate language UNIFORM was developed. A low-level denotational semantics was initially provided [Lan91s], 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 [Lan91d, LB90a, Lan91w]. Finally, a transformational semantics, by reduction of UNIFORM to $Z^+$, was produced [Lan91k], based on this axiomatic semantics, and formed the basis for the reverse-engineering of COBOL via transformation to UNIFORM and specifications.

The concept of a ‘process’ in UNIFORM was recognised 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 [BL91a, BL92a], and to tools for generating UNIFORM procedural code from $Z^+$ specifications [Lan90c].

3 Validation and Verification

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

- Verification of existing systems;
- 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. 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. In general reverse-engineering efforts have not achieved the extraction of this level of information, being concerned only with overall design and graphical aids to code comprehension. However situations arise in practice in which detailed functionality is essential to the understanding and re-engineering of the code. This arose even in a case study of a library database system, 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 [Lan91ab, Lan91z, Lan91j] contain more details of the tools.

3.1 Verification of Existing Systems

The problem addressed by this work was the support of formal verification activities on procedural (UNIFORM) source code. By verification we mean determination of the conditions under which a piece of source code satisfies a formal specification of its behaviour, and establishment of these conditions. The approach taken was 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 aspect of our work was the use of heuristics to generate plausible loop-invariants and intermediate assertions [LB90a]. The tools are designed to support interaction with the user at points where full automation is not possible, thus enabling the programmers intuitions about procedural code to be used. A limited theorem-proving and algebraic simplification capacity is built in to the system, using a decision procedure for Presburger arithmetic.

Definition

The notation \( \{ P \} \text{cmd} \{ Q \} \) means

\[
P \Rightarrow \text{wp}(\text{cmd}, Q)
\]

for predicates \( P, Q \) and command \( \text{cmd} \).

The method involves asserting a desired postcondition \( Q \) for a given subsection \( \text{cmd} \) of a program, and using the tool to generate successive preconditions, working in reverse execution order through the code, until a precondition \( P \) for \( \text{cmd} \) with respect to \( Q \) is obtained. If \( \text{cmd} \) contains loops, then further proof conditions will be generated as a result of the hypothesising of loop invariants. We cannot guarantee that \( P \) will be the weakest possible precondition, ie, \( \text{wp}(\text{cmd}, Q) \), but any complete execution of \( \text{cmd} \), commencing in a state satisfying \( P \), will produce a state satisfying \( Q \).

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

3.2 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, use an automatic non-interactive approach. We have chosen the latter approach, as it reduces the work required by the user, whilst not substantially decreasing the power of the refinements. A large example of application of our system, to a radar track-former, is given in [Lan90c]. 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 has been pointed out in [Coh89]; in that we can never in fact completely formalise 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 unformalised and partial.

Data and functionality are refined to procedural code in separate processes. The transformation
of data types from abstract forms such as sets, functions and sequences, into concrete types,
files, arrays, records, is the basis of refinement, in that this data refinement mapping determines
the mapping from specifications of programs into code. The interpretations of data types were
selected automatically, using the following transformation:

\[
\text{seq } t \rightarrow \text{array}[1, \ldots, +\infty] \text{ of } \tau(t) \\
t \rightarrow s \rightarrow \text{file of } r \text{ selectable by key } key1
\]

where \( r \) is a new record type:

\[
r :: \text{record} [key1 : \tau(t); key2 : \tau(s)]
\]

and similarly for partial functions (which are all assumed to be finite, although of arbitrary size,
as UNIFORM allows files and arrays to exist without specific size bounds). Schema types are
transformed to corresponding record types.

### 3.2.1 Procedural refinement

For the most part procedural refinement of specifications follows automatically from the data
refinements. When possible we combine assignments into concurrent assignments, and refine
non-deterministic forms of conditional into deterministic ones. This means that a specification
that uses schema disjunction:

\[
S_1 \quad Decs \\
\quad Pre_1 \\
\quad Op_1 \\
S_2 \\
\quad Decs \\
\quad Pre_2 \\
\quad Op_2
\]

\[
S \equiv S_1 \lor S_2
\]

is expanded out by pre-processing into

\[
S \\
\quad Decs \\
\quad (Pre_1 \land Op_1) \lor (Pre_2 \land Op_2)
\]

and becomes a procedure whose command is:
IF E1 THEN Cmd1 ELSE IF E2 THEN Cmd2 END IF END IF

where E1 is the refinement of Pre1, Cmd1 of Op1, and so forth.

Set-comprehensions, quantifications, and binding operators such as μ are translated into iterations performing searches over the given bounded domains. The systematic treatment of these is one of the largest parts of the system. The other critical aspect is the introduction of local variables for such iterations and other hidden operations involved in Z predicates.

A final warning is that we cannot expect the implementation to be perfectly correct with respect to the specification or even the abstract axiomatic semantics of the code as provided by [Lan91d] and [Lan91q], as the way in which (for instance) arithmetic expressions are evaluated may vary, and the range of defined numbers may vary. For similar reasons real numbers have been excluded from the Z subset.

3.2.2 Implementation results

The system consists of three parts: a parser for Z; a procedure compiling an internal Prolog form of the specification from the parse tree; and the actual refinement routines which interactively convert this form into UNIFORM code.

In total these comprise about 2,500 lines of Prolog, most of which is concerned with the third stage. The second stage involves the expansion of all schema references and types into corresponding sets of declarations or record types, the third stage performs the refinement of the types of the specification into UNIFORM types, and prompts for confirmation of these choices. At this stage the user can choose to implement sequences or sets as queues (say) as opposed to arrays, but this may not always be sensible (for instance if random access to the elements of the sequence is needed), and warnings will be given in the next stage if these make an algorithmic refinement impossible.

The algorithmic refinement stage is automatic. An internal Prolog form of the command will be created and stored as a fact in the database, and a visual representation is printed.

For example, in the track-former specification, the initialisation of the state of the specification:

\[
\begin{align*}
\text{Initial} \\
\text{STATE1'} \\
\text{ph'} = \langle \rangle \\
\text{unused'} = \langle \rangle \\
\text{th'} = \langle \rangle \\
\text{tracknumber'} = 0
\end{align*}
\]

becomes the parameterless procedure \text{Initial} with body

\[
\begin{align*}
\text{ptr\_ph} &:= 0; \\
\text{ptr\_th} &:= 0; \\
\text{ptr\_unused} &:= 0; \\
\text{tracknumber} &:= 0
\end{align*}
\]

The \textit{OptimalExtensions} schema
becomes:

PROCEDURE optimalextensions(
    IN: p:IndexMapsSet;
    OUT: p:IndexMapsSet;
    INOUT: !*** all of the state ***!);

OWNS
    ext : IndexMapsSet;
    ptr_p : INTEGER
BEGIN
    ptr_p := 0;
    DO VARYING ext IN p :
        IF function3
            THEN
                ptr_p := ptr_p +1
                p(ptr_p) := ext
            ELSE
                SKIP
        END IF
    END DO
END

where function3 tests if ext satisfies the condition in the body of the set comprehension; this involves a further iteration through p.

The system has also been extended to deal with object-oriented specifications in the Z++ [Lan91ae] language as its input, and supports manipulation of these specifications into refinable forms.

4 Reverse-Engineering of Existing Applications

Current practice in reverse-engineering or restructuring is concentrated on the extraction of graphical or statistical information from source code, which is useful for detecting design or coding flaws, but does not provide an explicit description of the functionality of the code, which is a critical feature if the code is to be reused. We will describe techniques which have proved effective in extracting both design and functionality from source code, and which have been applied to real
data-processing applications in industrial use. The techniques use object-oriented specifications as a unified representation for design and abstracted functionality.

Our techniques were developed to provide more powerful methods of information extraction and manipulation on applications. Instead of remaining at the code level, concise and high-level abstractions are derived from the application code, together with more conventional information, such as data-flow and control-flow diagrams.

Other approaches to the use of formal methods in reverse-engineering are those of Basili et al [BAC91], based upon a low-level and intensive validation of code, using classical Hoare-style assertions, and of Ward et al in the REFORM project [WCM89], which is based upon transformation of assembly language code into a wide-spectrum language. The work of Linger also utilises functional abstraction and program transformation [LHPH90], and that of Zimmer, which utilises object classes [Zim90], are also developed approaches. Our approach has some aspects in common with these, and integrates functional abstraction, language transformation and the formation of object classes with a formal specification language as its basis.

In contrast to the approach of [BAC91], we do not require detailed proof and validation procedures at the code level, but at the more easily manipulable abstraction level. The code-level validation approach was initially adopted in the REDO project, but abandoned for this reason, that the usability of the approach by maintenance engineers not familiar with formal methods was poor.

4.1 An Overview of the Method

The process of transformation and code comprehension that we have developed can be summarised as follows:

**Stage 1:** The COBOL program is translated into the intermediate language UNIFORM, and redundant statement structures are eliminated. MOVE X TO Y statements become assignments which involve a format conversion or casting: \( Y := f(FX,FY,X) \) where FX is the declared format of X, FY of Y. The relationships between data are translated into logical invariants of the program.

**Stage 2:** 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 TOs and other unstructured code constructs are eliminated.

**Stage 3:** 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.

4.2 The Reverse-Engineering Process

4.2.1 Stage 1: Translation from COBOL to UNIFORM

The reverse-engineering system translates COBOL code into UNIFORM [SC90] code, with additional information, such as the initial values of variables, being generated and stored, so that the resulting code is semantically equivalent to the original. In the process redundant constructions in COBOL are eliminated.
4.2.2 Stage 2: Higher level abstraction

The objects of the COBOL program are identified by examination of the FILE SECTION and ENVIRONMENT DIVISION entries in a program, from which we obtain the files of the program and their associated variables. Data-flow analysis of the UNIFORM translation of the program is performed, and this allows us to identify variables which are logically associated with the main data structures of the program, as well as the global data-flows between the main variables. We consider files, indexed arrays and reports to be such main variables, since they can be represented as objects with several attributes. We try to identify global functions which update or modify the main structure represented by the particular object class we are working on. Each such data flow identifies a candidate operation that updates the class, and we will initiate a more detailed analysis in order to find the precise functionality and the meaning of this operation.

Further techniques break up the functionality of the program into manageable chunks. We attempt to split the program into phases, where a phase is a maximal logically connected piece of code which contains no OPEN or CLOSE statements. In other words, it represents an execution period in which the files are processed in a particular mode, and hence should represent a single input-output function on these files. The modalities of the files during a phase gives us the form of this function, and the details are extracted from the code of the phase by means of functional abstraction, described below. If useful phases cannot be identified we use program slicing on the variables of an object $C$ and those variables we have identified as likely to be used in a method of $C$, to extract the changes to these variables from irrelevant functionality.

4.2.3 Functional abstraction

We assume now that we have identified suitable sections of code for further analysis. This code still contains the PARAGRAPHS and GO TO’s of COBOL, and retains the fall-through semantics that complicates reasoning. Therefore we need to simplify and reduce still further the number of code constructs. In our functional language, there are just two essential constructs: functional composition and conditional expressions: $\text{cond}(e, a, b)$, also written as $e \rightarrow a, b$, which return value $a$ if $e$ is true, $b$ otherwise. From these, and other expressions, we can represent the functionality of any derived program as a series of equations defining a set of functions, together with the declarations of the UNIFORM program.

It is necessary to capture the total functionality of an application if re-engineering is to be performed, since if the client wishes to retain all the existing functionality of an application, this must be re-created in the new version. More generally, areas where the old and new system differ in functionality should be exactly recorded.

4.2.4 Stage 3: Simplification of abstractions and design

Having obtained the equations, we can then perform a series of transformations and simplifications on them. A set of normalising transformations exist [BL91a], and these can be applied after every high-level modification to produce a consistent and standard form for the user. The equations can be manipulated and combined in various ways:

1. Substitution of a function body for its calls;
2. Eliminating recursion by recognising a simple iteration;
3. Adding in new equations, either as properties to be proved, or as assumptions to simplify the description;
4. Using invariants or other equations to substitute one expression for another inside a function description;

5. Renaming a function;

6. Deleting a function;

7. Proving properties of a function.

We can prove that these transformations will always enable us to rewrite a program into a structured form [LH91d]. In the following sections, we illustrate the complete procedure.

4.2.5 Example: Reverse-engineering a scientific application program

In this section we show how a scientific programming application may be reverse-engineered using the techniques outlined above, along with some higher-level reasoning in the abstraction domain. The example chosen is a technically difficult one, but it illustrates the techniques well. We start with the code A in Figure 1, introducing the labels shown at left.

```plaintext
init();
read m; read n;
E: do
   do
      A: if m=0 --LABEL A
         then n=n+1; break(1); --JUMP out of loop
         elsif n=0
            then push(1); m=m-1; n=1;
            else push(0); n=n-1;
         fi;
      done;
   done;
R: if empty() --LABEL Reentry
   then break(2); --JUMP two loops
   fi;
   pop(d)
   if d=0 then push(1); m=m-1; break(1);
   else m=m+1;
   fi;
   done;
return(n)
```

Figure 1: Source code A

The first thing one has to do with this code is restructure it. And the first part of this operation is the re-expression of subroutine calls as explicit functions (without hidden side-effects).

We introduce an explicit argument s to represent the stack of integers which is manipulated by the subroutine push (which takes a single integer argument and pushes it onto the front of the stack) and the subroutine pop (which takes a reference as argument and pops the top value off
the stack into the address supplied by the reference). For example, `push` becomes re-expressed as a function of type

```plaintext
int stack: push(int,int stack)
```

so that one will write `s=push(x,s)` instead of `'push(x)'`. Similarly, `empty` and `init` become functions of type

```plaintext
bool empty(int stack)
int stack init()
```

As part of this procedure, one has to replace call-by-reference semantics with the equivalent call-by-value semantics. This means replacing the subroutine call `'pop d'` with the function call `'(d,s)=pop(s)'`.

Restructuring then proceeds with the removal of all `break`'s. After restructuring, the code shown in Figure 2 is obtained, and three explicit functions are obtained as the semantics of the labeled segments of the original code. These are shown in Figure 3(a).

```
E: init(); read m; read n;
A: until(m=0) do
    if (n=0) then s=push(1,s); m=m-1; n=1;
    else s=push(0,s); n=n-1;
fi;
done;
n=n+1;
R: until(empty(s)) do
    (d,s)=pop(s);
    if (d=0) then s=push(1,s); m=m-1; goto A;
    else m=m+1;
fi;
done;
return(n);
```

**Figure 2: Restructured source code A**

We then simplify the mathematical functions by normalising the equations. This means

- eliminating the low-level intermediate equations for `d`, `s'` and `n'`. It is also convenient to
- use a more concise notation for the `push`, `pop` and `init` functions (equivalent in fact to introducing the underlying structures on which stacks are built). We
  - adopt the notation `'(x : s)'` or just `x : s'` in place of `'push(x,s)'`. Then
  - one can then write `'(d : s') = s'` in place of `'(d, s') = pop(s)'`, because the latter is equivalent to testing if `push(d, s') = s'`. And so we can replace an equation of the form `foo(s) = bar(d, s')...where (d, s') = pop(s)` with `foo(d : s') = bar(d, s')`. We also
\[
\begin{align*}
e(m, n) & = n' \\
& \text{where } (m', n', s') = a(m, n, \text{init})
\end{align*}
\]

\[
\begin{align*}
a(m, n, s) & = r(m, n + 1, s) & \text{if } m = 0 \\
& = a(m - 1, 1, \text{push}(1, s)) & \text{if } n = 0 \\
& = a(m, n - 1, \text{push}(0, s)) & \text{otherwise}
\end{align*}
\]

\[
\begin{align*}
r(m, n, s) & = (m, n, s) & \text{if } \text{empty}(s) \\
& = a(m - 1, n, \text{push}(1, s')) & \text{if } d = 0 \\
& = r(m + 1, n, s') & \text{otherwise}
\end{align*}
\]

\[
\begin{align*}
& \text{where } (d, s') = \text{pop}(s)
\end{align*}
\]

(a) Initial specification – translate literally.

\[
\begin{align*}
e(m, n) & = a(m, n, [\ ])[2] \\
a(m, n, s) & = r(m, n + 1, s) & \text{if } m = 0 \\
& = a(m - 1, 1, s) & \text{if } n = 0 \\
& = a(m, n - 1, s) & \text{otherwise}
\end{align*}
\]

\[
\begin{align*}
r(m, n, [\ ]) & = (m, n, [\ ]) \\
r(m, n, d : s') & = a(m - 1, n, s') & \text{if } d = 0 \\
& = r(m + 1, n, s') & \text{otherwise}
\end{align*}
\]

(b) Simplified specification – use structures instead of calls and eliminate local definitions.

\[
\begin{align*}
e(m, n) & = a(m, n, [\ ])[2] \\
a(0, n, s) & = r(0, n + 1, s) \\
a(m, 0, s) & = a(m - 1, 1, s) \\
a(m, n, s) & = a(m, n - 1, s)
\end{align*}
\]

\[
\begin{align*}
r(m, n, [\ ]) & = (m, n, [\ ]) \\
r(m, n, 0 : s') & = a(m - 1, n, s') \\
r(m, n, 1 : s') & = r(m + 1, n, s')
\end{align*}
\]

(c) Final specification – replace guards with pattern-matches.

Figure 3: Stages in the production of a literal specification for source code A
write ‘init()’ as ‘[ ]’, and consequently the test ‘empty(s)’ as ‘s = [ ]’, because it is equivalent to asking if ‘s = init()’.

That gives one the specification shown in Figure 3(b). One then normalises each individual equation. Particularly one substitutes occurrences of x by foo in equations guarded by x = foo.

This gives one the specification in Figure 3(c). Notice that the ‘otherwise’ in the equation for r must mean ‘if d = 1’ as the stack is only ever loaded with 1’s and 0’s. Higher reasoning is required to simplify this presentation.

4.2.6 Example cont’d: Reverse-engineering in the abstraction domain

It is apparent that function a just calls itself continuously until m = 0, when it calls r. So it is sensible to invent a new function a1 which represents the action of a up to and just before it calls r. We write a(m, n, s) = r(a1(n, m, s)) and eliminate r from the three equations for a by cancelling it where it appears on both sides. The result is shown in Figure 4(a), where the changes are indicated in boldface.

The mathematical justification (it needs justifying) for the elimination of r is that at this point we are only looking for some solution of the equations, not necessarily all of them. As the source program is deterministic, if we find a solution by this means, then it will be the uniquely correct one.

It is now apparent that a1 always delivers the result m = 0, so that we can, instead of writing ‘a1(m, n, s)’ always write ‘(0, a2(m, n, s))’, letting a2 stand for the function which delivers n and s alone. Simplifying, this gives one the specification shown in Figure 4(b).

Furthermore, only the second element of the triple delivered by r is ever required, so we can slim things down a little by replacing the equations for r(, , ) with equations for r2(, , ) = r(, , [2]) throughout, as shown in Figure 4(c).

4.2.7 Example cont’d: Data equivalences in reverse-engineering

At this point one has gone as far as possible without applying any intelligence. What comes next must rely somewhat on inspiration because the equivalence of different programs is not something which can be checked or discovered by a machine (an automatic procedure) acting on its own. Notice that whenever a 1 is pushed onto the front of the stack (1 : s appears on the right of an equation), the variable m is decremented, and whenever a 1 is popped off the stack (1 : s’ appears on the left of an equation) the variable m is incremented. We hypothesise that the stack is really just acting as a store for past values of m. Knowing the current value of m and how many 1’s are on the stack allows us to reconstruct all the old information.

Imagine a parallel stack t which contains these old values. The diagram 5 shows the relationship between t and s. From stack t and the current value of m, one can always reconstruct s (and m). The formal relationship expressed in this data equivalence is as follows (skip to the next paragraph for an explanation):

\[
\begin{align*}
(m - 1, 1 : s) & \equiv (m' - 1, t) \quad \text{if} \ (m, s) \equiv (m', t) \\
(m, 0 : s) & \equiv (m, m' : t) \quad \text{if} \ (m, s) \equiv (m', t) \\
(m, [ ]) & \equiv (m, [ ]) 
\end{align*}
\]

Each clause of the equivalence states what happens in one representation when a change occurs in the other. The first line says what happens if one pushes a 1 onto stack s and simultaneously
\( \text{Isolate the calls in } a \text{ prior to } r \text{ as function } a_1. \)

\[
\begin{align*}
e(m, n) &= r(a_1(m, n, [1]))[2] \\
a_1(0, n, s) &= (0, n+1, s) \\
a_1(m, 0, s) &= a_1(m-1, 1 : s) \\
a_1(m, n, s) &= a_1(m, n-1, 0 : s) \\
r(m, n, [1]) &= (m, n, [1]) \\
r(m, n, 0 : s') &= r(a_1(m-1, n, 1 : s')) \\
r(m, n, 1 : s') &= r(m+1, n, s')
\end{align*}
\]

\( \text{(a) Isolate the calls in } a \text{ prior to } r \text{ as function } a_1. \)

\[
\begin{align*}
e(m, n) &= r(0, a_2(m, n, [1]))[2] \\
a_2(0, n, s) &= (n+1, s) \\
a_2(m, 0, s) &= a_2(m-1, 1, 1 : s) \\
a_2(m, n, s) &= a_2(m, n-1, 0 : s) \\
r(m, n, [1]) &= (m, n, [1]) \\
r(m, n, 0 : s') &= r(0, a_2(m-1, n, 1 : s')) \\
r(m, n, 1 : s') &= r(m+1, n, s')
\end{align*}
\]

\( \text{(b) Notice that } a_1 = (0, \ldots) \text{ always, and re-express in terms of } a_2. \)

\[
\begin{align*}
e(m, n) &= r_2(0, a_2(m, n, [1]))[2] \\
a_2(m, n, s) &= \ldots \\
r_2(m, n, [1]) &= n \\
r_2(m, n, 0 : s') &= r_2(0, a_2(m-1, n, 1 : s')) \\
r_2(m, n, 1 : s') &= r_2(m+1, n, s')
\end{align*}
\]

\( \text{(c) Notice that only the second component of } r(\ldots) \text{ is needed.} \)

Figure 4: Reasoning about the specification to achieve mathematical simplification

13
Figure 5: The relationship between stacks $s$ and $t$

decrements variable $m$. The change is from a stack with shape $s$ to one with shape $1 : s$, and from a value $m$ in $m$ to a value $m - 1$ in $m$, and it is represented by the

$$(m - 1, 1 : s) \equiv \ldots \text{ if } (m, s) \equiv \ldots$$

part of the statement. The corresponding change in the other representation is that the stack $t$ is unchanged – the decrement in $m$ cancels the external visibility of the ‘push 1’ because $t$ only records the sum of $m$ and the number of 1’s on $s$ – but the value in $m$ is still seen to decrease, from $m'$ to $m' - 1$. Using the value $m'$ instead of $m$ ensures that we do not prejudice (wrongly?) that these values are always the same. The contribution to the equivalence statement is

$$\ldots \equiv (m' - 1, t) \text{ if } \ldots \equiv (m', t)$$

and putting the two parts together gives the clause of the data equivalence.

This allows us to rewrite the equations as shown below (the relevant instantiations of the data equivalence which have been used to obtain each equation are shown to the right of that equation).
The last equation shows that \( r_2 \) depends only on \( n \) and \( t \), not on \( m \). The recursion is towards an (unstated) boundary condition when \( m \) is as big as the value at the front of stack \( t \).

The reason for considering this data equivalence in the first place is the observation that all the original equations preserve the sum of \( m \) and the number of 1’s in the group at the head of stack \( s \) across the ‘\( = \)’ sign. This value is plainly an important quantifier of the state of the system, and it is the value always held at the back of the stack \( t \). The next value from the back is the sum of \( m \) and the number of 1’s in the first two groups (separated by a zero) at the head of \( s \), and so on.

Because of \( r_2 \)’s dependence only on \( n \) and \( t \), one can consider \( r_2 \) to be never called except with \( m = 0 \), so it is useful to give a name to \( r_2(0, n, t) \): we replace it with \( r_0(n, t) \) everywhere, and dispense with the last equation above.

Another data equivalence is needed to unravel this set of equations: we can see that \( a_2 \) pushes values on the stack, and \( r_0 \) takes them off again later, but only in order to call \( a_2 \) immediately with the popped value as (an) argument. So the stack just serves as a reservoir of arguments to \( a_2 \) for use when it is executed again later. What we can do is give \( k = r_0(\_, t) \) as an argument to \( a_2 \) instead of \( t \), writing ‘\( a_2(m, n, k) \)’ instead of ‘\( a_2(m, n, t) \)’, and in the clause where \( a_2 \) finally returns the pair \( (n + 1, t) \) for use by \( r_0 \), have it instead execute \( r_0(n + 1, t) \) directly by writing ‘\( a_2(0, n, k) = k(n + 1) \)’ instead of ‘\( a_2(0, n, t) = (n + 1, t) \)’. The function \( k \) is called a continuation function.

We carry out this procedure below. (Technically, this is an operation of generalisation on the equations, because we recover the original equations by setting \( k \) to be the function \( \nu \mapsto (\nu, t) \) for any particular fixed \( t \).)

<table>
<thead>
<tr>
<th>( e(m, n) )</th>
<th>( = )</th>
<th>( r_0(a_2(m, n, [_])) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a_2(0, n, t) )</td>
<td>( = )</td>
<td>( (n + 1, t) )</td>
</tr>
<tr>
<td>( a_2(m, 0, t) )</td>
<td>( = )</td>
<td>( a_2(m - 1, 1, t) )</td>
</tr>
<tr>
<td>( (m, s) \equiv (m, t) )</td>
<td>( (m - 1, 1 : s) \equiv (m - 1, t) )</td>
<td></td>
</tr>
<tr>
<td>( a_2(m, n, t) )</td>
<td>( = )</td>
<td>( a_2(m, n - 1, m : t) )</td>
</tr>
<tr>
<td>( (m, s) \equiv (m, t) )</td>
<td>( (0 : s) \equiv (m, m : t) )</td>
<td></td>
</tr>
<tr>
<td>( r_2(m, n, [_]) )</td>
<td>( = )</td>
<td>( n )</td>
</tr>
<tr>
<td>( r_2(m, n, m': t) )</td>
<td>( = )</td>
<td>( r_2(0, a_2(m' - 1, n, t)) )</td>
</tr>
<tr>
<td>( (m, s) \equiv (m', t) )</td>
<td>( (m + 1, s) \equiv (m + 1, t) )</td>
<td></td>
</tr>
<tr>
<td>( r_2(m, n, t) )</td>
<td>( = )</td>
<td>( r_2(m + 1, n, t) )</td>
</tr>
<tr>
<td>( (m, 0 : s) \equiv (m, m': t) )</td>
<td>( (m, 1 : s) \equiv (m, t) )</td>
<td></td>
</tr>
</tbody>
</table>

\begin{align*}
\text{Another data equivalence is needed to unravel this set of equations: we can see that } a_2 \text{ pushes values on the stack, and } r_0 \text{ takes them off again later, but only in order to call } a_2 \text{ immediately with the popped value as (an) argument. So the stack just serves as a reservoir of arguments to } a_2 \text{ for use when it is executed again later. What we can do is give } k = r_0(\_, t) \text{ as an argument to } a_2 \text{ instead of } t, \text{ writing ‘} a_2(m, n, k) \text{’ instead of ‘} a_2(m, n, t) \text{’, and in the clause where } a_2 \text{ finally returns the pair } (n + 1, t) \text{ for use by } r_0, \text{ have it instead execute } r_0(n + 1, t) \text{ directly by writing ‘} a_2(0, n, k) = k(n + 1) \text{’ instead of ‘} a_2(0, n, t) = (n + 1, t) \text{’. The function } k \text{ is called a continuation function.}
\end{align*}
Moreover, we can abstract away the stack argument \( t \) from the equations by writing \( r_0 \) as a curried function, \('r_0(n)(t)'\) instead of \('r_0(n, t)'\), and generalising \( a_2 \) slightly by writing \('a_2(m, n, f)(t)'\) instead of \('a_2(m, n, f(t))'\).

One may generalise a function \( a(\_\_\_) \) in any of several ways:

- Replace a subexpression \( \text{foo} \) which appears on the rhs of the equations defining \( a \) with a new variable \( x \), and declare \( x \) to be an extra parameter to \( a \).
- If \( a \) often appears in equations with a compound expression \( \text{foo} \) as parameter, as in \('a(\text{foo})'\), and \( \text{foo} \) uses a subexpression \( x \) repeatedly, then one may write it as a function : \( a(e, x) \), where \('e(x)'\) can be instantiated to \( \text{foo} \). This introduces a function as (an extra) parameter \('e'\), as well as the ordinary parameter \('x'\).
- One may drop one of the equations which governs the function.
- One may replace any subexpression \( \text{foo} \) on the rhs of the defining equations with a generalisation (this is a recursive definition!).
- One may explicitly replace any expression on the rhs of the governing equation with \('\bot'\) (‘don’t know’).

The second option is the course followed here! We have \('\text{foo}'\) as \( f(t) \), and we introduce \( f \) and \( t \) as separate parameters to \( a_2 \). Then we cancel the \('(t)'\) where it appears on both sides, in \('a_2(0, n, k)(t) = k(n + 1)(t)'\), for example:

\[
\begin{align*}
e(m, n) &= a_2(m, n, r_0(\_, [\ ])) \\
a_2(0, n, k) &= k(n + 1) \\
a_2(m, 0, k) &= a_2(m - 1, 1, k) \\
a_2(m, n, k) &= a_2(m, n - 1, k') \\
\text{where } k'(\nu, t) &= k(\nu, m : t) \\
r_0(n, [\ ]) &= n \\
r_0(n, m' : t) &= a_2(m' - 1, n, r_0(\_, t))
\end{align*}
\]

The two steps, which we have justified with proper mathematical reasoning above, can be carried through in a single step by using the data equivalence below – ‘data is equivalent to function which encodes data’:

\[
\begin{align*}
e(m, n) &= a_2(m, n, r_0(\_, [\ ])) \\
a_2(0, n, k) &= k(n + 1) \\
a_2(m, 0, k) &= a_2(m - 1, 1, k) \\
a_2(m, n, k) &= a_2(m, n - 1, k') \\
\text{where } k'(\nu, t) &= k(\nu, m : t) \\
r_0(n, [\ ]) &= n \\
r_0(n, m' : t) &= a_2(m' - 1, n, r_0(\_, t))
\end{align*}
\]
\[(m' : t) \equiv k' \quad \text{if} \quad t \equiv k\]
\text{where} \ k' \nu \tau = k \nu (m' : \tau)

The assertion is that the stack of shape \(t\) is equivalent to some function \(k(\_, \_)\) (parameterised as \(k(\_, \_)\) in the equivalence) which takes an index \(\nu\) and a reference stack \(\tau\) as its parameters. If one changes \(t\) to \(m' : t\), then this is reflected in a change from \(k(\nu)\) to \(k'(\nu)\) (for each given \(\nu\)), where \(k'(\nu)\) acts on the reference stack \(\tau\) as \(k(\nu)\) would have acted on a reference stack \(m' : \tau\). If one first synchronises \(t\) and \(\tau\) at the stage when they are both empty, then \(k\) has encoded into it precisely the manner in which \(t\) is composed at any point. This is tricky to visualise, but it is really just an algebraic device to make the following sequence of mathematical transformations work.

Now we set out the few final steps to the answer. Looking for a solution, we assume that \(a_2(m, n, k) = k(A(m, n))\) for some function \(A\), and so we get the equations in Figure 6(a).

The idea is that a real computer does not pass functions about – so one should be able to eliminate the function \(k\) if \(a\) is computable. Here, letting \(RA(m, n)\) stand for \(r_0(A(m, n))\), and replacing \(k(\_) (t)\) with \(r_0(\_) (t)\) throughout, we get the equations in Figure 6(b).

Then, setting \(t\) to \([\ ]\) throughout (since this is the normal initial state of the stack) and making use of the equation \(RA(m, n)[\ ] = A(m, n)\), we get the description in Figure 6(c), which shows that \(e(m, n)\) is precisely the Ackerman function \(A(m, n)\), since these are its defining equations.

### 4.3 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 variables. The user selects which variable he wishes to examine, and the slice of the functional abstraction on this variable is calculated (so that some functions which do not affect the variable are discarded). The entity life history diagram for this variable is then calculated from the normalised functional abstraction [LH91d].

### 4.4 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. [Lan91t] 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 parameterised object classes [Lan91k]. 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
\[
e(m, n) = r_0(A(m, n)) \[ \]
\]
\[
k(A(0, n)) = k(n + 1)
k(A(m, 0)) = k(A(m - 1, 1))
k(A(m, n)) = k'(A(m, n - 1))
\]
where \( k'(\nu t) = k\nu(m' : t) \)
\[
r_0(n)[ ] = n
\]
\[
r_0(n)(m' : t) = r_0(A(m' - 1, n))(t)
\]

(a) Assume \( a_2(m, n, k) = k(A(m, n)) \).

\[
e(m, n) = RA(m, n)[ ]
\]
\[
= A(m, n)
\]
\[
\{ r_0(n)[ ] = n \}
\]
\[
RA(0, n)(t) = r_0(n + 1)(t)
\]
\[
RA(m, 0)(t) = RA(m - 1, 1)(t)
\]
\[
RA(m, n)(t) = r_0(A(m, n - 1))(m : t) \quad \{ r_0(n)(m : t) = r_0(A(m - 1, n))(t) \}
\]
\[
= r_0(A(m - 1, A(m, n - 1)))(t)
\]
\[
= RA(m - 1, A(m, n - 1))(t)
\]

(b) Instantiate \( k(\_)(t) \) to \( r_0(\_)(t) \) throughout & rename \( r_0(A(\_, \_)) \) as \( RA(\_, \_) \).

\[
e(m, n) = A(m, n)
\]
\[
A(0, n) = n + 1
\]
\[
A(m, 0) = A(m - 1, 1)
\]
\[
A(m, n) = A(m - 1, A(m, n - 1))
\]

(c) Instantiate \( t \) to \( [ \ ] \) throughout and use \( RA(m, n)[ ] = A(m, n) \).

Figure 6: Mathematical reasoning on the equations describing code A.
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.

4.4.1 Example: Design abstraction

One of the largest systems to be processed by our system was a suite of programs from the early 1970’s [Peg91], which implemented a library database. This consisted of some 50,000 lines of code in total, although functional abstraction was restricted to smaller sections of code, with manual application of the methods also being used. 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 parameterisation 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;
- Parameterisation of operations.

Classes associated with each of the three main database files BOOK-FILE, COPY-FILE and TITLE-STACK-FILE are recognised. 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 formalised from the documentation, and was verified using transformations on abstractions. Thus, we have the outline class:

```plaintext
CLASS Library_Database
  OWNS
  BOOK-FILE : x13 \rightarrow ddata \times \{\'0\', \'1\}\;
  COPY-FILE : x13 \times 0 .. 99 \rightarrow cdata \times \{\'0\', \'1\}\;
  TITLE-STACK-FILE : x50 \times 999 \rightarrow x13 \times \{\'0\', \'1\}\;
  OPERATIONS
    DROP : CARD \rightarrow ;
    CREATE : seq CARD \rightarrow
  INVARIANT
    Inv.Library
END CLASS
```
In addition, there are two other classes, a Card_File_Class for the input stream and an Error_Class which handles the exception conditions. The type $x_{13}$ is the set of sequences of 13 characters, and in the above, variables of this type hold ISBN numbers. The type $x_{50}$ will hold book titles.

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

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

4.5 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 [Lan91v], which removed all occurrences of the concurrent execution operator (||).

5 A Specification-based Approach to Maintenance

In this section of our work, we defined 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 [LH91g].

The components of the method are:

- a standard specification style for systems;
- exact semantic correspondences between code and specification forms [LB90a];
- 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 industry in using and learning Z, we felt such a formalism should be adhered to, especially as it is widely regarded as one of the best or most usable formal specification languages [Phi90].

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. Unlike Z, we take the B-tool convention for specifying operations of classes; that is, in the predicates defining class operations a default assumption of state preservation is made; if no explicit change is made to variable $x$ then it is assumed not to change over the operation. In conventional Z even identity changes $x' = x$ need to be explicitly stated unless they are logically implied by the explicit statement of the operation. 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 [Lan91d].
5.1 The Z++ Language

Only a brief description will be given here. An overview is given in [Lan91ae], and [LH91b, LH91a] provide details of the semantics. Methods of refinement and reasoning about Z++ specifications are given in [LH91f], and application examples in [BL91a, BL92a, LBH91a, LH91i].

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

Definitions of global types

Definitions of object classes

<table>
<thead>
<tr>
<th>STATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables used in every operation</td>
</tr>
<tr>
<td>Global invariants on these variables</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Initial</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATE'</td>
</tr>
<tr>
<td>Initial values of variables</td>
</tr>
</tbody>
</table>

... Definitions of operations

That is, a specification is a sequence of paragraphs of formal text (possibly 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 [LBH91b].

A specification describes a hierarchy of type and object class definitions, in which a class $C$ inherited by class $D$ occurs higher in the hierarchy. Another important relationship between classes is that of refinement: A class $D$ refines a class $C$ if $D$ can implement all of $C$’s operations in such a way that the interpretation of $C$’s operations by $D$ refine the original definitions (in the usual procedural sense [MRG88]). In our semantics for the language, where classes are denoted by logical theories, refinement corresponds to relative theory interpretation, similar to the concept of a morphism in the institutions used by Goguen for object-oriented languages [Gog91].

Another hierarchy is determined by the client relationship [Mey90] between classes: A class is used as a client by another class if it is used as the type of an attribute of this class, and then its operations can be used as method calls in the definitions of the methods of this class. Because the state of a class is encapsulated, and can only be changed or viewed from the outside via the methods of the class, such a hierarchical organisation promotes separation of concerns: a development team can work separately on different parts of a specification, provided they guarantee that the operations of the classes they define meet the requirements of those classes which use them. The internal details of the classes are irrelevant from the point of view of functionality.

This also allows changes to be made to classes at one level of the usage hierarchy without necessarily requiring changes to the levels which they use or which use them. A change in a
user-interface for example would not normally require any change in the layers beneath it, and so forth. Indeed work in WP5 of the REDO project applied the concepts of object-orientation to the extraction of user-interfaces from applications and their redesign [Smi91].

An example of a class from the marriage base specification [LH91g] is:

```
CLASS mclass
  OWNS
    contents : Name ↔ Name
  RETURNS
    MEMBERS : → P Name
  OPERATIONS
    ADDMARR : Name Name → ;
    REMM : Name Name →
  ACTIONS
    MEMBERS         ==>  ∪ contents;
    ADDMARR m f    ==>  contents' = contents ⊕ {m ↦→ f};
    REMM m f       ==>  contents' = {m} △ contents
END CLASS
```

This is a refinement of a previous version of this class, and this refinement required no changes in the level of the specification which used this class. Other notations, closer to Z, are also available as syntactic variants [LBH91b].

5.2 Metrics

Metrics of maintainability were developed for these specifications, based on the structural metrics defined in WP7 of the REDO project [Jac90]. These metrics favour modularisation of a specification into moderate-sized classes over monolithic specifications.

5.3 Development and Refinement

A transformational approach to refinement is advocated. The path the developer takes in passing from the specifications to the code is recorded by means of a tree of transformation laws and what sections of specification or code they were applied to. Specific class refinement laws, in a natural deduction proof style, are contained in [LH91f]. Ideally, such a ‘proof’ tree can be reused and understood in much the same way as mathematical proofs are intended to be. A proof of a mathematical result is retained because of the insight that it may yield in understanding the critical points of the result, and in obtaining analogous results.

5.4 Reusability

Reusability refers to the ability to reuse some aspect of a software development in other new developments. The reusable part is not necessarily restricted to code but may also include specifications, development histories and documentation. Much research has been undertaken in assessing program transformation as a means of aiding in the reusability of software components [ABFP86, Nei84, PS83, BM84, BP86]. The results suggest that a system based on transformations can provide an effective means of recording development histories and verification proof steps which may be revised and reused to provide alternative implementations. Thus the benefits of our approach also includes these advantages.
5.5 Summary

The maintenance method we have devised can be summarised as follows:

1. Creation of a formal specification, either from user requirements, or from a reverse-engineered application.

2. Development of the application (in the first case), using the system to record the development and refinement steps. In the second case abstraction transformations will have been recorded during reverse-engineering.

3. Translation of change requests on the application into change requests on the specification, using the links established in stages 1 and 2.

4. Implementation of these changes to the specification, with guidance for the user on the possible consequences of these changes throughout the system.

5. Regeneration of the application implementation from the changed specification, together with revised documentation.

Formal specifications are the central item of information here, bridging the gap between the informal but comprehensible user requirements, and the over-detailed formal description that is the source code. It is the level at which changes can most easily be effected.

5.6 A Formal Model for Maintenance and Development

A model of the maintenance and development process was also specified in $Z^{++}$ [Lan91p], so providing a formal description of the proposed development and maintenance system outlined above.

Conceptually, a software module or component can be considered to have several attributes:

1. A user requirement for its behaviour;

2. A specification of its behaviour;

3. A development history, a sequence or tree of refinements beginning with 2. and including 4.:

4. A current refinement,

together with logical relationships between these: that the specification satisfies the requirement and that the descendants of a node in the development history refine their ancestor. There are also characteristic operations, such as:

1. Apply a refinement rule to the current refinement;

2. Build a specification;

3. Select a new current refinement (i.e., start from a new direction in refining the system),
together with several others, which we will want to perform on the module.

These attributes and operations are the basis of the meta-level class *Software* used in our
development and maintenance system. In development we will typically begin with an empty
development history, and only a user requirement, and work to fill in the other details; in reverse-
engineering we will have one or possibly more 'leaf nodes' of the development tree, and we will
want to work backwards to a useful specification and requirement (description) of the system.

The items, such as specifications, used as attributes of the *Software* class are themselves
objects, with attributes such as *author*, *date*, *level* and so forth, and can be manipulated in
their own right, and linked to several separate *Software* components. The object style thus gives
great freedom and flexibility in manipulating and recombining system descriptions: all we need
to know about an object instance is its name, and we can then incorporate it as an attribute of a
suitable higher-level object, subject to type constraints. For instance, in reverse-engineering of a
program, we may derive information, such as a data-flow graph of the code, that we will reuse in
different several operations - such as the recognition of outline objects in the code, or in giving a
data-complexity metric to the code, and we need to possess this information as a freely available
entity, in addition to its use as an attribute of the *Code* type.

The *Software* class is defined as follows:

```
CLASS Software
  OWNS
    requirement : FormalText;
    specification : Class;
    developmenth : tree[Code];
    currentcode : Code
  INVARIANT
    (∀ specification) ⇒ meaning(requirement)
    ∀ x : (nodes developmenth) ∀ y : descendants x • y refines x
    (root developmenth) refines specification
    currentcode ∈ (nodes developmenth)
  RETURNS
    Ancestors : Code → seq Code;
  OPERATIONS
    Refine : Law → ;
    InitiateR : FormalText → ;
    InitiateS : Class →
  ACTIONS
    Ancestors a ===> ancestors(developmenth, a) ;
    InitiateR req ===> INITIATE requirement WITH req ∧
      INITIATE_NULL specification ;
    InitiateS spec ===> INITIATE requirement WITH spec ∧
      INITIATE specification WITH spec ∧
      INITIATE (root developmenth) WITH spec ∧
      developmenth = { (root developmenth) → }
    Refine law ===>
      currentcode' = law OF currentcode
      Add_Descendant currentcode' currentcode TO developmenth
END CLASS
```
The set Law of refinement laws is the set of method identifiers used in refinement in the Code class. Each of these refinements needs certain parameters, which will be asked for as part of an interactive process initiated within the Code class. At this level all we need to know is the name of the refinement operation. A method call INITIATE requirement WITH req is an abbreviation for an assignment requirement' = INITIATE(requirement, req).

Another advantage of the object style of design is the enhanced security and maintenance of internal consistency of descriptions: we cannot change the attributes of a class instance in any way other than by its supplied methods, which may in addition require that certain conditions are satisfied before they can change the class instance. In particular we cannot change the root of the development history without changing the specification, in an instance of the Software class.

An outline of the Code class is as follows:

```
CLASS Code
OWNs
cobolcode : CCode;
uniformcode : UCode;
funcabstraction : FAbs;
dataflow : WeightedGraph
OPERATIONS
InitiateC : CCode -> ;
InitiateU : UCode -> ;
CobolToUni : -> ;
Abstract : -> ;
Dataflow : ->
ACTIONS
CobolToUni ==> uniformcode' = convert_cobol_to_uniform(cobolcode);
Abstract ==> funcabstraction' = abstraction(uniformcode);
Dataflow ==> dataflow' = data_view(uniformcode)
END CLASS
```

In addition there are methods which perform refinements of the uniformcode attribute.

At the global level, there is a class which manages the user interaction; it possesses attributes currentS : Software, currentC : Code, and so forth, and, given a user request to perform a particular high level operation, it selects the appropriate current instance of Software, Code or Class, and the appropriate methods to apply to this instance to carry out the high-level operation.

### 5.7 Formalising Class Transformations

We can express transformations of code and specifications as methods of the appropriate class. Class transformations, such as the addition of a new attribute, can be explicitly defined:

\[
v \notin \text{dom } signature \land \text{Add}_\text{New}_\text{Attribute } v \ t \implies \text{signature}' = \text{signature} \cup \{(v, t)\}\]

as methods of the class:

```
CLASS Class
OWNs
```

25
Similarly with more complicated refining operations, such as the replacement of a method by a more refined method.

As an example, we can consider the development of the ‘students who do exercises example’ [LBH91b] to be carried out in this framework, as the successive application of rules to transform an abstract set of requirements through an algebraic specification, to an abstract state-based specification, a concrete state-based specification, and a procedural pseudo-code description. We have the following graphical presentation of the refinement history of this Software instance:

The transformational paradigm for development [Bau85, Bur91, Nei91] has been followed and expressed as the application of methods of object classes representing software and specification components. Other approaches, such as the reuse paradigm of Basili [BC90, BC91] could be incorporated into the system, simply by allowing more types of software class, and providing operations which return metrics of reusability from these classes.

Similar representations can be made for reverse-engineering [Lan91p].

5.8 Laws

Formalising the transformation of software allows us to write down axioms and general properties that are satisfied by these transformations, and these can be verified using a precise semantics. For instance, we have that the operations of adding distinct attributes can be performed in either order, with the same result:

\[
\forall C : \text{Class}; v, w : \text{Ident}; t, s : \text{Ident} \mid v \neq w \bullet \\
\text{Add\_New\_Attribute}(v, t, \text{Add\_New\_Attribute}(w, s, C)) = \\
\text{Add\_New\_Attribute}(w, s, \text{Add\_New\_Attribute}(v, t, C))
\]

Similarly, if a method \( m \) does not mention an attribute \( v \), then adding \( m \) after \( v \) yields the same class as adding \( v \) after \( m \).
6 Decompilation Techniques

The Dictionary of Computing [Ill90] gives the following (abridged) definitions for a compiler and decompiler (or reverse compiler):

**compiler**  A program that translates high-level language into absolute code . . .

**decompiler**  A program that attempts to . . . translate back from machine code to something resembling the source language. The task is difficult and not often attempted.

This section outlines how a decompiler may be constructed given only a knowledge of or access to the specification for a forward compiler, and some facility with Prolog [CM87] or other high-level languages.

6.1 Decompilation

A compiler may be specified by a description of how each construct of the source language is translated into a sequence of object code instructions [Hoa91]. It is possible to produce a compiler prototype almost directly from such a specification, and this is quite a usual route for the construction of a compiler nowadays – first the specification is written, and then a compiler-compiler such as the UNIX yacc tool [AJ74] reads the specification and writes the code for the compiler. But the specification can be read in the other direction too, allowing one to construct a decompiler from it, either using a decompiler-compiler [Bre90b] or working more directly.

A quicker route to a working compiler prototype than applying a full-blown compiler-compiler to the compiler specification is to simply encode a version of the compiler specification in a declarative language such as Prolog [War80]. This should be executable immediately, if reasonable care is taken. Writing the compiler in Prolog is a key idea in producing a decompiler here: in theory a Prolog program can be run backwards, if certain provisos are met, because the source code has no procedural directionality [Llo87]. Each Prolog program declares a relationship between input and output variables, but ‘input’ and ‘output’ can be switched round if the user calls the program in a slightly different way. The rest of this section is devoted to briefly explaining the idea.

The trick to making a compiler in the form of a logic program [BHP90, HHBP90], is to read the compiler specification as defining a relation between valid high-level and low-level program constructs. Normally a high-level program is supplied as input to a compiler and object code is returned, but what is wanted here is to supply object code and return the equivalent high-level program(s), resulting in a decompiler, and the switch-round will be entirely external to the Prolog code. In practice, however, some alterations to the program will have to be made: the clauses of the forward compiler logic program will have to be reordered to ensure termination when it is run ‘backwards’, and in particular, all the arithmetic clauses will need to be reversed from ‘x = y + z’ to ‘z = y − x’, (or be written as reversible relations ‘plus(x, y, z)’). The reordering is necessary because of Prolog’s very simple depth-first and left-to-right search strategy. In actual use, it will probably be necessary to make judicious use of the cut pseudo-clause (‘!’) as well for efficiency reasons.

Compiler specifications may be presented in Prolog as a three-way relation \( C \) between high-level code \( p \), the matching object code \( m(s : f) \) (lying between a start address \( s \) up to but not including a finish address \( f \)), and a symbol table of addresses \( \Psi \) which lists the memory locations (used in the object code) against each name (used in the source code) [BHP90, Hoa91, HHBP90]:

27
\[ C(p, m[s:f]), \Psi) \]

For example, the clause of \( C \) which says that ‘the compilation of the sequential composition of two programs \( p \) and \( q \) depends on compiling the two subprograms contiguously in memory’ derives from the following specification statement:

**Specification \{sequential composition\}**

If \( s \leq j \leq f \)
and \( C(p, m[s:j]), \Psi) \)
and \( C(q, m[j:f]), \Psi) \)
then \( C(p; q, m[s:f]), \Psi) \)

Note that this specification introduces an intermediate address \( j \) between start address \( s \) and finish address \( f \). During forward compilation, the code \( p \) will be compiled first and since the start address \( s \) is known, its finish address \( j \) can be calculated (it is just \( s + n \) where \( n \) is the length of the compiled code segment produced from \( p \)). Then, compiling \( q \), knowing the start address \( j \) results in the final finish address \( f \) being calculated in the same way. This can be coded in "standard" Prolog [CM87, Qui90] as follows:

\[
\text{c}(P;Q,M/S:F,Psi) :- \\
c(P,M/S:J,Psi), \\
c(Q,M/J:F,Psi), \\
\text{ensure}(S=J=\leq F). \\
\]

Here the ensure clause checks that \( s \leq j \) and \( j \leq f \). (Actually this is superfluous in the forward direction because negative length sections of the address table \( m \) will never be generated.)

For decompilation, the object code \( m[s:f] \) will be known, but the high-level program \( p; q \) will be unknown. The logic program must ensure that the unknown intermediate address \( j \) ranges between \( s \) and \( f \), and some reordering of clauses is necessary because of Prolog’s naive left-to-right depth-first search of clauses during execution. The ‘ensure’ term must come first in order to generate plausible values for \( j \):

\[
\text{cr}(P;Q,M/S:F,Psi) :- \\
\text{ensure}(S=J=\leq F), \\
\text{cr}(P,M/S:J,Psi), \\
\text{cr}(Q,M/J:F,Psi). \\
\]

So some clever encoding of the ensure clause (and other arithmetic clauses) is necessary when using standard Prolog. However, the newly emerging field of constraint logic programming [Coh90, Col90] – which allows a variety of constraints (including numerical constraints) to be added to a logic program – allows the clause ensure \( S=\leq J=\leq F \) to be encoded as a simple constraint \( \{S=\leq J, J=\leq F \} \), and then the same program may be used for both forward and reverse compilation without alteration. So if a reversible language is used to encode a compiler, then it can be used as a decompiler too!

A complete forward and reverse compiler for a simple imperative language, including recursion, has been produced as a prototype in Prolog using this technique [Bow91b], and the use of constraint logic programming languages looks promising for the future. Decompilation is only useful, of course, if the object code is available and it is required to reproduce the higher level ‘original’ as an intermediate stage to aid understanding or restructuring. The decompiled code can then be abstracted to a specification by other techniques developed by the REDO project [LB91d, LB90a].
6.2 Decompiler Compilers

Another approach to constructing a decompiler from the compiler specification is to arrange for a decompiler-compiler tool [Bre90b] to read the specification and write the code for the decompiler. Again, it is advantageous to use a high-level language (such as Prolog or a functional language), but not completely necessary. It is only required that the language allow the use of subroutines as parameters.

The concept behind a decompiler-compiler is that the clauses of the specification for the compiler can be read as instructions on how to explicitly enumerate all the source codes which could give rise to a given object code. The following is the appropriate version of the specification to work from:

**Reverse Specification** {sequential composition}

\[ C(\_m|s:f),\Psi) \text{ contains } p \mid q \]

where \( j \) is from \( [s...f] \)

and \( p \) is from \( C(\_m|s:j),\Psi) \)

and \( q \) is from \( C(\_m|j:f),\Psi) \)

and what it describes is a set (unary relation) \( C(\_m|s:f),\Psi) \) which contains the source code construct \( p; q \) under certain conditions. The conditions are that \( p \) and \( q \) are to be found in the sets \( C(\_m|s:j),\Psi) \) and \( C(\_m|j:f),\Psi) \) respectively. Imagine that we want to write a procedure which enumerates all the elements of these sets in a list structure. It could look like:

```c
list function rce(m,s,f,Psi){
  local j,p,q,r
  r := EmptyList
  for j from list ListFromTo(s,f) do
    for p from list rce(m,s,j,Psi) do
      for q from list rce(m,j,f,Psi) do
        r := AppendToList(r,Seq(p,q))
  return(r)
}
```

and all a decompiler-compiler has to do is construct this code automatically from the specification given.

The only difficulty arises from multiple clauses in the specification. The full compiler specification from which this example is taken also contained the clause

**Reverse Specification** {abort}

\[ C(\_m|s:f),\Psi) \text{ contains } ABORT \]

and the proper approach is to

1. write an enumeration function for each alternate clause of the specification (‘sequential composition’ and ‘abort’ above), and

2. define the enumeration function for the whole specification as the concatenation of the enumerations of the clauses, thus:

```c
list function rce(m,s,f,Psi){
  return(ConcatenateLists(rce0(m,s,f,Psi),rce1(m,s,f,Psi))
}
```
Here $rce_0$ is the enumeration function associated with the ‘sequential composition’ clause, and $rce_1$ is that associated with the ‘abort’ clause:

```haskell
list function rce1(m,s,f,Psi) {
local r
r := EmptyList
r := AppendToList(r, Abort())
return(r)
}
```

In fact this method of constructing a decompiler only ‘works’ if there are only a finite number of source code constructs which could have given rise to a particular object code. If there are an infinite number, the program may find itself attempting to append one more element to the end of an already infinite list, which will bring a long pause to the procedure!

The way out of the difficulty is to replace the `ConcatenateLists` function with an ‘interleave’ function, which does not place the elements of the second list after all the elements of the first list, but instead interleaves them together as fairly as possible. But one also has to ‘interleave’ the `for x from list y` operations too, and it is simplest to do this by using an explicit list-valued function instead of the two-deep `for...list...loop`, thus:

```haskell
list function rce0(m,s,f,Psi) {
local j,p,q,r
r := EmptyList
for j from list ListFromTo(s,f) do
  for (p,q) from list XLists(rce(m,s,j,Psi), rce(m,j,f,Psi)) do
    r := AppendToList(r, Seq(p,q))
return(r)
}
```

and here `XLists` does the complicated interleaving of the product of the two lists in a fair way. For a discussion of the general computational problem and the complete decompilation algorithm, see [Bre90b]. The method given here is expensive on space if the substrate language does not use some form of ‘garbage-collection’ to reclaim abandoned list structures. Even with garbage collection the method is expensive on time because the structures are at least made before being abandoned, and the problem then is that the substrate language is eager to evaluate them even though they will not ultimately be needed. In a lazy language like Haskell or Orwell [BW88], however, these structures will never be completed and the time-behaviour will be better. We can simulate the action of a lazy language by using a slightly different concept as the basis of the decompiler technique, and the remainder of this section will deal with that.

Laziness can be simulated by using `objects` instead of lists. The objects are procedures which when given the `NEXT` command, respond with the next item in the corresponding list, and when given the `FIRST` command give the first element in the corresponding list, and reset their internal counter, and we use an object to stand in for the set $C(\_, m[s,f], \Psi)$. The decompiler can construct the object for the ‘abort’ clause of the specification as follows:

```haskell
class com(s,f)->rce1(m,Psi) {
case com in
  NEXT) return(0)
  FIRST) return(Abort())
}
```
meaning that the response to a `FIRST` element of list' request is always the `ABORT` code construct, and to a `NEXT` command the response is always `sorry, already finished` (here signalled by a zero). The class being takes the variables $m$ and $\Psi$ as parameters because these do not vary significantly. Since a class is a generic object, one will create an object $o$ with these parameters already set when one calls `$o := rce1(m,\Psi)$', and one has to send it its remaining parameters with the message:

```plaintext
NEXT(s,f)->o
```

The distinction between classes (generic pattern for an object) and objects is not particularly important in this case, because there is no internal state. But the object constructed from the `sequential composition` clause of the specification does give rise to an object with internal state (represented by the `persistent` local variables $*j, *p, *q$) however, and because of the possibility of inadvertently sharing internal states, the object must set up distinct private copies $rce1$ and $rce2$ of the $rce$ enumerators at initialisation, because these are required to count through their enumerations independently of each other. This is handled by the declaration of $rce1$ and $rce2$ as being instances of the class $rce(m,\Psi)$ at the top of the file, along with the other local variables. Which $*j, *p, *q$ one gets is intended to be dependent on the message parameters $s$ and $f$, and this is handled by shifting the variables address to the correct place in an array.

The final object (see Figure 7) looks complicated, but is easy to construct automatically. Further alterations are required to make it count in an interleaved way, and some extra objects are still manufactured and discarded, but the overall semantics is closer to call-by-need (lazy evaluation) than in an imperative language with lists.

### 6.3 Conclusions

In practice the object code from an optimising compiler will be considerably more complicated than any of the examples which have so far been attempted. However, the Prolog approach does have the advantage that extra clauses of the compiler specification will be accommodated without affecting the existing clauses.

The efficiency of the Prolog decompilers generated so far is not great, but decompilers produced by the decompiler-compiler are more efficient. And when multiple answers are possible, the first answer returned will normally be of greatest interest (and the program can be structured with this in mind), so it is not necessary to search through them all (e.g., all the possible combinations of sequential composition).

Additionally, the layers of the source code language can be treated separately. For example, individual subroutines can be recognised and isolated early on in the decompilation process, so that smaller blocks of code may then be decompiled separately. Assuming that subroutines are all approximately of the same order of magnitude in size, the computational complexity is then linear in the number of subroutines. Indeed, if parallel computation is available, separate subroutines could be decompiled in parallel, thus reducing the computation time still further – essentially to a (large) constant time.

The decompiler-compiler approach is based on functional rather than relational concepts. Because of the irreversible nature of functions when there are implemented on a machine, compilation and decompilation must be handled separately. A different set of concerns arise with this approach, particularly with regard to the enumeration of sets of possible high-level programs or object codes to ensure that the required programs are enumerated fairly.
class com(s,f)->rce0(m,Psi) {
  persistent array j[*,*], p[*,*], q[*,*]
  persistent object rce1, rce2 of class rce(m,Psi)

  j:=&j[s,f]; p:=&p[s,f]; q:=&q[s,f] -- instantiate j, p, q

  case com in
    NEXT) if (*q := NEXT(*j,f)->rce2) -- stepped q
      then return(Seq(*p,*q)) -- return p; q
      then return(0) -- q empty, give up!
    if (!(*q := FIRST(*j,f)->rce2) -- reset q
      then return(0) -- q empty, give up!
    if (*p := NEXT(s,*j)->rce1) -- finished q, stepped p
      then return(Seq(*p,*q)) -- return p; q
      if !(*p := FIRST(s,*j)->rce1) -- reset p
        then return(0) -- p empty, give up!
      if (s<=+++j<=f) -- finished p, stepped j
        then return(Seq(*p,*q)) -- return p; q
      return(0) -- finished j, give up!
    FIRST) *j := s
      if !(*p := FIRST(s,*j)->rce1)
        then return(0) -- p empty, give up!
      if !(*q := FIRST(*j,f)->rce2)
        then return(0) -- q empty, give up!
      return(Seq(*p,*q))
  esac
}

Figure 7: Code for the final object
6.3.1 Future problems

The problems of parsing has so far been ignored by using the abstract syntax directly. Fortunately, this can be made fairly readable using modern declarative programming languages since infix, prefix and postfix operators with specified associativity and precedence are normally allowed. However, in practice a real compiler would need a parser (and a static semantics checker). A decompiler would require a concrete syntax to be generated from the abstract syntax produced by the decompiler presented here. Luckily this is far easier that the reverse procedure since only correct abstract syntax is generated and thus error checking is not necessary. All these extra phases are possible in Prolog, or some other high-level declarative language, although perhaps not as efficiently as other approaches [Paa91].

The development of techniques of inductive logic programming [MF90], which aim at the derivation of Prolog programs from examples and background knowledge, could provide the possibility of automatically synthesising the compiling specification program from examples of triples \((p, \Psi, m)\) of an input source program \(p\), symbol table \(\Psi\), and the compiled object code \(m\). This then obviates the need for (perhaps unfounded) assumptions on the part of the decompiler writer about the semantics of the source code.

7 Conclusion

The REDO project has been active for three years from January 1989 to December 1991. The project has been involved with software maintenance in general, and reverse engineering in particular. At Oxford we have concentrated on the formal aspects of the work. In addition, we have produced a number of prototype tools, mostly based around Prolog. A significant number of external publications have resulted directly from the work on the project: these, together with the internal project documents produced at Oxford, are listed at the end of this report. The internal documents have been deposited at the Oxford University Computing Laboratory library for future reference.

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), and Giorgos Papapanagiotakis, formally at CTC (Athens), have co-authored a number of reports and publications.

Bibliographies

Please note that copies of REDO project documents and PRG Technical Reports are available from the Librarian at the PRG. (See title page for address.)

A REDO Publications and Reports, Oxford


Peter T. Breuer. Breaking and building higher-order operating systems (and some comments on Church’s Thesis). REDO project document 2487-TN-PRG-1024, Oxford University, UK, February 1990.


Peter T. Breuer. Higher order operating systems as a computational domain. REDO project document 2487-TN-PRG-1028, Oxford University, UK, March 1990.


Peter T. Breuer. REDO intermediate language; requirements analysis. REDO project document 2487-TN-PRG-1029, Oxford University, UK, January 1990.


Kevin C. Lano and Howard Haughton. Extracting design and functionality from code. REDO project document 2487-TN-PRG-1085, Oxford University, UK, December 1991.


B Other References


<table>
<thead>
<tr>
<th>Reference</th>
<th>Author(s)</th>
<th>Title</th>
<th>Publisher/Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC90</td>
<td>Stanley-Smith C., Cahill A.</td>
<td>UNIFORM: A Language Geared to System Description and Transformation</td>
<td>University of Limerick, Ireland, 1990.</td>
</tr>
</tbody>
</table>


