The Art of Computer Un-Programming: Reverse Engineering in Prolog

Peter T. Breuer[1]

Oxford University Computing Laboratory, 11 Keble Road, Oxford OX1 3QD, UK.
Email: Peter.Breuer@comlab.oxford.ac.uk

Abstract. A suite of Prolog tools for reverse-engineering and validating COBOL programs has been developed as part of the ESPRIT REDO project [**?**]. These tools produce functional abstractions, object-oriented designs and documentation from raw source code, with the aim of improving comprehensibility and maintainability, and this article discusses the tools and aspects of their programming.

1 Reverse-Engineering and Validating COBOL

The COBOL language [**?**] hinders program comprehension in several ways:

1. by its use of formats instead of types (which means that each assignment between variables usually involves an implicit conversion of data from one format to another);
2. by the mixed use of `PERFORM` statements and the ordinary *fall-through* execution of paragraphs;
3. by the inclusion of unstructured constructs such as `GO TO`'s and `ALTER GO TO`'s (which establish computed aliases for labels in the code); and
4. by excessive proliferation of specialised verbs and variants of verbs, the exact semantics of which in turn requires very specialised knowledge on the part of the programmer.

These features of COBOL make recognising reusable program components difficult, and therefore mean that most extant COBOL code is difficult to maintain and difficult to integrate into an evolutionary business strategy. Many software houses today find themselves with products inherited from an earlier era that are comprehensible only to the original development team, who may well have left the company. When a modification or an improvement to the functionality is required, the code has to be either jetisoned or laboriously researched, and it is in response to this problem that the ESPRIT II project REDO [1] [**?**] was launched in 1989.

This project has sought to develop methods of maintaining and improving old Fortran and COBOL applications, and has met with some success. The

[1] "REengineering, DOcumentation and validation of systems", ESPRIT project no. 2487.

project partners as a whole early on came to the view that reverse engineering was the key technology to investigate, and in this article such methods and tools for reverse engineering are described as have been developed at Oxford as part of the REDO project.

These tools take in COBOL code and produce *specifications* through an interactive process supported by considerable automatation. The specifications are not exact transcriptions of the code, but deliberate *abstractions* away from the programming detail, produced by engineers engaged in documenting, decomposing and analysing the functionality of an application code. The Oxford tools, in distinction to contributions from other partners in the project and, so far as we know, attempts at reverse engineering worldwide,

- produce specifications that are written in a recognised formal specification language (Z [**?**]), and
- are arranged, structured and presented in object-oriented packages of varying levels of abstraction,

These outputs satisfy the growing requirement of many contractors for

- formal specifications and
- a recognised top-down develop method

as part of the contracted code, in order to guarantee future maintainability and keep down the cost of service arrangements.

The objects and classes which come out of the analysis provide reusable software components worthy of inclusion in a development software library. Library-based development also supports the drive towards lower-cost production of software, and may also be insisted upon in future by software consumers interested in the quality and efficiency of their suppliers.

The Oxford tools are unique in attempting to capture the *functionality* of the code, not merely the formal arrangement of its control-flow, the data-flow, or the types of data structures (all these views are available from the toolset too).

The features of COBOL pointed out above, and the correctness requirements on the formal specifications which have to be produced, make software tools and a properly understood *formal method* for going about reverse engineering a necessity. It is here that Prolog has proved most useful, because we have been able to give methods phrased at the level of code-to-code relationships (between one formal language and the same or a different language) and transcribe them into Prolog quite quickly. Moreover, the Quintus Prolog [**?**] environment has supplied a satisfactory front end to the prototype toolset, and the compiled code has operated efficiently, and, more importantly, convincingly, in actual trials on medium sized applications. In additon, we have been able to integrate theorem provers and tools which validate the code against the derived specifications within the whole ensemble. AI techniques generate heuristics which aid the user in guessing suitable specifications in those cases which are not automatic, And we are also

able to interpret some of the derived specifications themselves within Prolog in order to obtain executable simulations which are useful in visualization.

Below, the *process* followed by the toolset is described.

On loading, the application source code is parsed and automatically transformed into an internal intermediate language, UNIFORM [**?**], which is somewhat simpler than COBOL. This is in order to facilitate later operations. All the tools work from this representation of the application, which is maintained in a persistent database [**?**] and is available to the user during certain operations. An external database is used for reasons of compatibility with the rest of the project developments, but a Prolog database would have been satisfactory otherwise. An example of UNIFORM is given in Figure **??**(b), and it can be seen to correspond almost exactly to the the original COBOL in Figure **??**(a), with the addition of proper scoping constructs (`begin`, `end`) in order to permit the use of local variables and private functions.

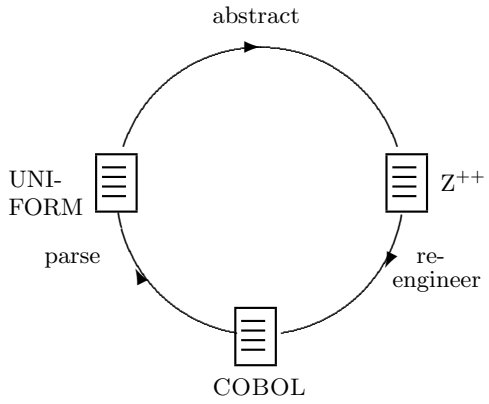

Fig. 1. The re-engineering cycle.

The parser library in the Quintus environment has been most useful in enabling us to construct a parser where no single-token lookahead parser would have sufficed. The parser is superior in speed to a version constructed using the standard UNIX *lex* and *yacc* tools [**?**] by other partners in the project.

The semantics of UNIFORM has been completely formally specified, so the transformation apparently succeeds in giving an unambiguous formal semantics to COBOL (!) but it should be borne in mind that formal specifications need not be fully deterministic, and in fact many different implementations will satisfy the specification used. The tools work on the logic of the common abstraction.

*Outline object*designs are first suggested to the user by heuristics which identify significant subsections of the code, using the data-flow graph for guidance (this