Use of Code Synthesis in ZEUS
Use of the Gödel Metasynthesizer

P. M. Hallam-Baker
Oxford University Nuclear Physics Lab
Deutsches Elektronen Synchrotron

Gödel, a tool for building code synthesizers is described together with its application to code used within the ZEUS experiment. Code synthesis allows code to be written in much shorter timescales than traditional means and produces code of commercial quality.

1 The Role of Code Synthesis.

The “waterfall” paradigm of software design contains three sequential stages, specification, implementation and validation. The practical disadvantage of this paradigm is that mistakes made in the implementation and specification stages may not become apparent until the final implementation stage. This leads to the requirement that each of the stages be an iterative process such that both implementation and specification are continually refined. In this paradigm errors in the specification stage require the implementation stage to be repeated several times, even in the case that the implementation team perform their task correctly in every case. Such a system raises considerable questions concerning personnel management and morale besides the inefficient use of resources. In addition repeated specification changes are likely to lead to code which has lost it’s original clarity and consistency.

A code synthesizer automatically translates a specification into an implementation. Such systems may be either generic or highly specific in their range of application. Generic systems, usually based upon the use of functional languages in general suffer from performance difficulties which restrict their use to the design stage. Gödel [1] is a tool for producing code synthesizers targeted at a highly restricted area, thus allowing highly problem specific optimizations to be made while retaining most of the benefits of generic synthesizers. This permits a logical separation between an implementation team producing a synthesizer and a specification team providing an input file for it.

2 The Gödel Metasynthesizer.

The Gödel Metasynthesizer consists of two parts, the structure definition tool and the application generator. The structure definition tool produces a parser which parses the specification file to produce a parse tree which is passed to the application generator. This is then traversed by the application generator to produce the output file (figure 1).

Gödel makes no assumptions as to the language the translator output is to be written in. Gödel synthesizers outputting C, occam, yacc and UIL exist. Similarly use could be made of FORTRAN, C++, Smalltalk or a text formatter such as \LaTeX. Several Gödel applications make use of multiple output languages.
2.1 The Structure Definition Tool.

The Structure definition tool is used to define the underlying structures of the translator. Types may be either simple types (INTEGER, STRING, TEXT), type constructors (TYPE, LIST, ALT) or labels (LABEL, REFERENCE, ID). The following is an example structure file:

```
TYPE animal
  STRING name
```

2.2 The Specification File.

The specification file syntax is defined by the structure definition file. Indentation is used to delimit the start and end of lists of elements. If the structure file incorporates labels these are automatically resolved (in both forward and backwards directions). A specification file for the 'animals' example might be:

```
"goose"
"chicken"
"rabbit"
```

2.3 The Application Generator.

The application generator is loosely based on the C preprocessor and uses the # mark to introduce preprocessor controls. All elements apart from controls are written to the output file. Controls are divided into execution controls providing iteration over lists and conditional translation and output controls enclosed in angled brackets, which cause elements to be extracted from the parse tree and written to the output file.

This design allows the translator to be produced from existing code in a gradual manner. This allows Gödel to assist a code restructuring exercise whereby 'dusty deck' code which has become unmaintainable may be recovered.

Gödel permits elimination of repetitious code which for various reasons cannot be encapsulated within a subroutine, allowing the size of source files to be minimized. This is a significant advantage when a project might otherwise require several hundred thousand lines of code to be written and maintained manually.
An application generator script for the ‘animals’ example might be:

```plaintext
#FILE
#FOR animal a FROM toplevel.animal
A #<a->name#> is an animal.

#END FOR
#END FILE
```

This produces the following output:

A goose is an animal.
A chicken is an animal.
A rabbit is an animal.

The system might be used to produce code for text formatters instead of plain text:

A \textit{#<a->name#>} is an animal. \texttt{(ISpX)}
A \texttt{NEWTERM}(#<a->name#>) is an animal. \texttt{(SDML)}

3 Applications of Gödel.

3.1 A Transputer Communications Harness.

The Inmos Transputer is designed as a building block for producing MIMD processor arrays. Processes on adjacent computing nodes may communicate via point to point serial links. Separate provision must be made for other communications requirements. The harness provides communications support together with essential operating system features. The harness comprizes three code synthesizers:

- The Meta State Management (MSM) tool allows run control software to be developed in a manner which matches the finite state machine model of the specification.

- The Routing Kernel provides communication between arbitrary Transputer nodes with deadlock freedom guaranteed.

- The VMS server provides access to file and screen handling facilities on the VAX host.

3.2 The TROJAN User Interface Management System.

The development of user interface software is complex and time consuming. This is greatly simplified by the use of an user interface management system which hides implementation detail from the application. TROJAN produces interfaces which allow use of either X-Windows or character cell (SMG/curses) based user interfaces in a manner which is transparent to the application code.

Troyan user interfaces have been used widely within the central data aquisition system and many older systems are gradually being replaced. The TROJAN synthesizer itself makes use of two synthesizers which assist in code building:
• The coding of the application generator file for X-Windows Motif. This is intended as the core of a future system supporting use of other interface toolkits.

• The generation of the finite state machine upon which the character cell version of TROJAN is based

3.3 The Fast Graphics Language (FGL) 3D Data Viewer.

FGL is a 3D data viewing system which allows rapid generation of displays for event and other data. FGL incorporates a TROJAN user interface. The event display is fast enough to run on standard workstations without a graphics accelerator. Currently only use of the Phigs graphics library is supported, it is hoped to produce a GL version in the near future.

3.4 Gödel

In accordance with it’s name, Gödel is itself coded as a Gödel synthesizer.

4 Conclusions.

The code synthesis is a valuable tool in building reliable systems meeting complex and demanding specifications. A principal benefit is the clear separation it enforces between specification and implementation. Thus when requirements change, implementation effort is not wasted and when new architectural opportunities occur they may be exploited with minimal effort.

The greater consistency of the synthesized code leads to a much higher degree of reliability over conventionally produced code through the elimination of minor coding errors.

Several software systems based on the Gödel code synthesizer have been developed within the ZEUS collaboration. These cover user interface design, event viewing, data formatting, and a communications harness for transputer networks. These systems have run reliably over extended periods of time.

Acknowledgements

I would like to thank my supervisors, Robin Devenish and Ian McArthur for their invaluable help. Thanks are also due to Chris Youngman, Aaron Levy and many other members of the ZEUS collaboration who made this work possible. This work was funded by SERC, the Digital Equipment Corporation and DESY.

References