Decision Tree Organization for GUI Generation

Andrew Strelzoff
University of California Santa Barbara
Department of Computer Science
Santa Barbara, California 93106
strelz@engineering.ucsb.edu

Linda Petzold
University of California Santa Barbara
Department of Computer Science
Santa Barbara, California 93106
petzold@engineering.ucsb.edu

Abstract

An application's decision tree is a state machine which the user traverses, filling in input and making decisions, to reach a runnable state. In previous work we showed how a decision tree can be derived from an application's code by static semantic analysis [9]. This decision tree can be translated directly to a graphic user interface [GUI], using XML-to-GUI technology, but the resulting user interface has numerous redundancies and artifacts of the code from which it was derived. In this paper we describe an expert systems approach which produces a user interface that has a more natural organization which approaches the look and feel of an interface produced by hand.

Keywords

automated software engineering, user interface requirements, reverse engineering, scientific computing, XML technology, expert systems

1. Introduction

Automated Graphic User Interface [GUI] generation and maintenance is an important problem that has extensive applications for many genres of programming, including scientific computing. The problem is to enable relatively inexperienced interface programmers to generate and maintain sophisticated GUI's. Many physical scientists and engineers develop their own applications. These applications would be much easier to learn and use with a GUI front end. The developers of scientific applications typically lack the time and expertise necessary to produce and maintain high quality GUI's. A key step in the generation of a GUI is the derivation of the user interface requirements for the underlying application.

The derivation by hand of user interface requirements for complex scientific computing applications by hand is a non-trivial problem. The MAUI software developed at Sandia National Laboratories [3] takes an XML description of an application's user interface requirements and generates a GUI skeleton which is a useful platform for further development. As part of the MAUI project a senior postdoctoral researcher was given the task of producing user interface requirements for a medium-sized numeric package. This task required 6 months [2].

In previous work [9] we developed a reverse engineering process which derived user interface requirements for complex scientific computing applications. The output from this process was XML encoded to facilitate translation to a GUI using MAUI or another XML-to-GUI technology. The problem with this work was that although the resulting GUI was **valid** it did not have the natural organization of a GUI built by hand.

The problem is to develop a process which mimics the organizational tasks a human programmer performs when building a user interface. Our approach involves a sub-topic within the field of expert systems, the construction of decision tables. This area of research developed out of circuit design where it was desirable to build logically circuits with as few logical gates as possible. The Karnaugh map graphical method was developed for small problems [1]. We use the Quine-McCluskey method which was developed for larger problems [7]. We then follow the procedure outlined in Vanthienen and Wets [10] for producing optimal decision trees from decision tables to automatically organize the user requirements. The result, when translated to XML and then to a GUI, has more of the look and feel of an interface constructed "by hand".

In the next section we examine the process of producing and refining user interface requirements in more detail, but first we introduce our target application. DASSL: Differential Algebraic System Software [8] is a well-known software package developed by Petzold in 1982 to solve Differential Algebraic Equations [DAEs]. DAEs are, roughly speaking, systems of ordinary differential equations coupled with constraints. These systems arise naturally in the

simulation of a wide range of problems in science and engineering [4]. The DASSL package has evolved into DASPK [5] and then DASPK3.0 [6]. Over the years a great deal of functionality has been added, first for solving much larger systems of DAEs [5], and later for sensitivity analysis [6].

2. Methodology

Our method for generating well organized XML-encoded user interface requirements has 6 steps.

1. **(I) Characterization**: Every variable and subroutine is characterized by tracing usage throughout the program.
2. **(II) Graph Representation and Coloring**: The program is transformed into a directed graph. Each node in the program graph is determined to be in one of the following classes: {Decision Point, Requirements Block, Computational Block}.
3. **(III) Removal of Extraneous Information**: All computational blocks, loops and backjumps are removed. Error detection jumps are also removed. The result is a decision tree composed of decision points and user requirement blocks.
4. **(IV) Organizing the Decision Tree**: Each path through the decision tree is recorded as a logical expression. These expressions are reduced using the Quine-McCluskey algorithm. The reduced decision table is then turned into an optimized decision tree using the Vanthienen-Wets procedure.
5. **(V) Constructing the Decision Point Diagram**: The leafs of the decision tree are annotated with runnable states to form a decision point diagram [DPD].
6. **(VI) Generating an XML Description**: The decision diagram is then translated into XML.

Steps one through three are fully described in [9] so we will only briefly outline them here.

2.1 Characterization

Our approach is to find program branches which are controlled by user input. These are decision points. User input can be determined by tracing variable usage. Variables which are used but never initialized (or at most initialized very early, perhaps as a default value) must be user input. Variables which then appear only in control statements are assumed to be control variables. This set of branches in which control variables appear are control points.

We should note at this point that we are dealing with FORTRAN77 which was selected as a starting point because it is still important in the scientific computing community and because it is relatively simple to parse. Extension of this work to more complex languages such as C is discussed further in the the last section.

2.2 Graph Representation and Coloring

The next step is produce a network representation of the application. The procedure is similar to Basic Block derivation, which is a common compiler procedure. A Basic Block is a sequence of consecutive statements in which flow of control enters at the beginning and leafs at the end without the possibility of branching except at the end. The network is usually not a tree but it does have a root: the beginning of the primary subroutine and one leaf: the execution end-point of the primary subroutine.

After the derivation of the Basic Blocks we need to color the resulting network. Blocks which contain decision points are designated as decision blocks. Blocks which contain user input but which do not involve control variables are requirements blocks (the user must provide some data or subroutines if the branch which includes the requirement block is to be taken). All other blocks are computational blocks.

2.3 Removal of Extraneous Information

In this step all of the computational blocks are removed. When a block is removed, all parents of the block are connected with the removed block's child blocks.

The type of applications under consideration are not interactive, therefore the control variables are usually set early in the program and never changed. Thus, loops and backjumps which are effectively acting as loops will not figure in the solution and are removed.

Error jumps which take program flow either to the end of the program or to a diagnostic subroutine which then exits the program are also removed. As a result many branches will now have only one outgoing path.

The last block in the program is always the END statement. This is removed, with the result that we now have a tree-like network with a starting root and one or more leafs. We call this network a decision tree although it is not a proper tree since there may be joins where differing user options share the same code.

2.4 Organizing the Decision Tree

We begin by collecting paths from the starting point down to each leaf. There may be more than one path if there is a "join" where differing options share the same code. On

each path we collect all the branch statements and requirements blocks traversed as logical implicants. The set of all these logical statements is then submitted to the Quinne-McCluskey algorithm.

The Quinne-McCluskey algorithm generates all prime implicants. Given a logical expression $F(a, b, c, ...)$, a prime implicant of F is a product term $P(a, b, c, ...)$ which implies F, and which has the property that if any variables are removed from P the resulting expression will not imply F. The second step is to extract the minimum number of prime terms which form a cover for the original expression. This is accomplished in two steps. First all essential prime implicants are accepted. A essential prime implicant is one which covers terms not covered by any other prime implicant. This implies that the remaining terms are all covered by more then one prime implicant. For each remaining term the algorithm choses the "eclipsing" prime implicant. A implicant eclipses another implicant if it covers its terms. The result is a minimal cover of prime implicants which forms a minimal equivalent logical expression.

In the process we are describing result is a much reduced set of non-redundant requirements for each possible decision path. These results are collected in a "contracted" decision table.

There are three possible ways to construct a new decision tree from the contracted decision table as outlined in [10].

1. **Naive Balanced**:The decision table is transformed into a tree from left to right, with each column resulting in a new leaf. The result is a balanced tree.

2. **Minimal Node Balanced**:This procedure results in a tree with the fewest number of leafs.

3. **Optimal Unbalanced**:This procedure minimizes the average height of the resulting tree meaning the user will traverse the fewest possible states in the interface to reach a runnable state.

2.5 Constructing the Decision Point Diagram

The next operation is to add runnable states to the decision tree. The result is a decision point diagram [DPD]. DPDs are typically in one of the following forms.

1. **Simple.** No decision points were found so the simplest possible DPD is a default requirements block with an attached runnable state.

2. **Tree or "Connected Forest".** The most common pattern for a DPD of a complex application is a decision tree or a forest of decision trees connected only at the root default requirements.

3. **Complex Network.** Scientific computing applications are often nested or combined to form larger more complex applications. For example, the software package DASPK is sometimes used to generate consistent initial conditions and then restarted to solve the problem. This results in a DPD which takes the user through a decision tree to set up the initial conditions for the problem and then a second decision tree to set up the software to solve the problem.

Numeric software modules can be combined in extraordinarily complex ways. For now we concentrate on straightforward tree-like DPDs and relatively simple combinations like the example of restarting DASPK given above.

2.6 Generating an XML Description

The MAUI [3] software an XML-to-GUI engine for scientific computing has been developed at Sandia National Laboratories. Translating a DPD to MAUI-XML is relatively simple. Each decision point becomes a MAUI class, with the elements of the requirements block leading to that decision as member variables. Subsequent decisions are sub-classes of previous decisions. A simple example is given in figure 1. The MAUI-XML to generate the GUI in figure 1 begins as follows:

```
  <Maui RootClass="DDASPK">
<Class type="DDASPK">
 <Fields>
  <Double label="T" name="T"/>
  <Double label="TOUT" name="TOUT"/>
  <Double label="TSTOP" name="TSTOP"/>
 </Fields>
</Class>
<Class type="INFO1" base="DDASPK">
 <Fields>
  <Int label="INFO(1)" name="INFO1"/>
 </Fields>
</Class>
<Class type="INFO7" base="INFO1" label="INFO7">
 <Fields>
  <Int label="INFO(7)" name="INFO7" default="0"/>
 </Fields>
</Class>
<Class type="INFO16" base="INFO1" label="INFO16">
 <Fields>
  <Int label="INFO(16)" name="INFO16" />
  <Double label="DDAWTS" name="DDAWTS"/>
  <Double label="DINVWT" name="DINVWT"/>
   <Int label="NEQ" name="NEQ"/>
 </Fields>
</Class>
<Class type="INFO4" base="INFO16" label="INFO4">
 <Fields>
  <Int label="INFO(4)" name="INFO4"/>
 </Fields>
</Class>
```

Using the extensibility features of MAUI the input subroutines can be composed, compiled and run from within the MAUI environment.

3 Case Study

In the next section we outline the derivation of user interface requirements for one of our target applications, DASPK.

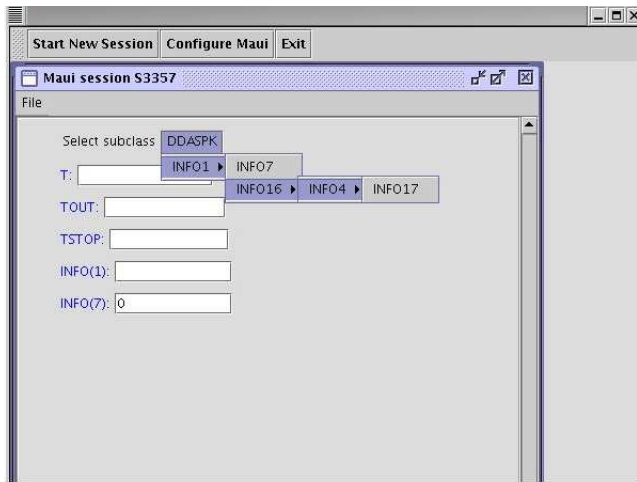

Figure 1. Maui GUI skeleton for central section of DASPK

4. Discussion

The remaining work for this research is to determine which of the three methods of decision tree organization (or some combination or other novel method of organization) produces the GUI with the most "look and feel" of one produced by hand.

We would also like to produce an extensive case study showing the advantages of the method chosen for one of our target codes.

5. Acknowledgments

This work was supported by grants: NSF/ITR ACI-0086061, NSF/KDI ATM-9873133, and DOE DE-FG03-00ER 25430.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison Wesley, 1974.

[2] B. Boggs. Conversation with Maui research team. Sandia/Livermore, 2002.

[3] P. Boggs, L. Lehoucq, K. Long, A. Rothfuss, E. Walsh, and R. Whiteside. A Maui user's guide. http://csmr.ca.sandia.gov/projects/maui/docs/MauiTutorial/.

[4] K. Brenan, S. Campbell, and L. Petzold. *The Numerical Solution of Initial Value Problems in Differential-Algebraic Equations*. 2nd Edition, SIAM, Philadelphia, 1996.

[5] P. Brown, A. Hindmarsh, and L. R. Petzold. Using krylov methods in the solution of large-scale differential-algebraic. *SIAM J. Sci. Comp.*, 15:1467–1488, 1994.

[6] S. Li and L. Petzold. Design of new Daspk for sensitivity analysis. Technical Report 1999-28, University of California Santa Barbara, 1999. www.cs.ucsb.edu/research/trcs/abstracts/1999-28.shtml.

[7] E. J. McCluskey. Minimization of boolean functions. *Bell Syst. Tech J*, 1956.

[8] L. Petzold. A description of Dassl: A differential/algebraic c system solver. Technical Report SAND82-8637, Sandia National Lab, 1982.

[9] A. Strelzoff and L. Petzold. Deriving user interface requirements from densely interleaved scientific computing applications. IEEE, 15th Automated Software Engineering Conference, 2003.

[10] J. Vanthienen and G. Wets. From decision tables to expert system shells. *Data and Knowledge Engineering*, 1997.