Deriving User Interface Requirements from Densely Interleaved Scientific Computing Applications

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

Deriving user interface requirements is a key step in user interface generation and maintenance. For single purpose numeric routines user interface requirements are relatively simple to derive. However, general numeric packages, which are solvers for entire classes of problems, are densely interleaved with strands shared and mixed among user options. This complexity forms a significant barrier to the derivation of user interface requirements and therefore to user interface generation and maintenance. Our methodology uses a graph representation to find potential user decision points implied by the control structure of the code. This graph is then iteratively refined to form a Decision Point Diagram, a state machine representation of all possible user traversals through a user interface for the underlying code.

Keywords

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

1. Introduction and Motivation

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 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]. This is a problem in reverse engineering. Considerable research has been done in this area [9, 10, 13, 11, 16]. The paper "Detecting Interleaving" by Rugaber, Stirwalt and Wills [17] lays out some useful terminology for dealing with reverse engineering computational codes. The authors describe a "plan" to denote a computational structure which is intended to solve a problem using a particular methodology. The "interleaving" of plans in an

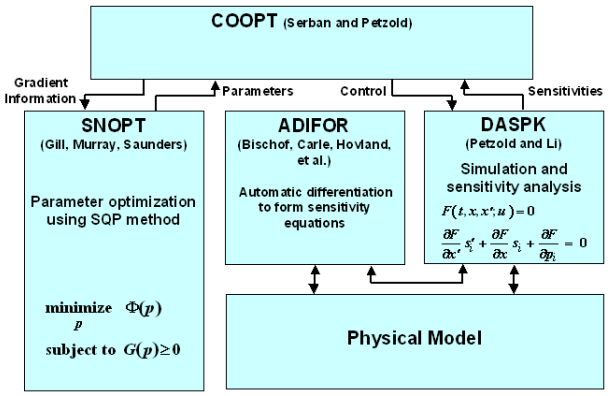

Figure 1. DASPK as part of COOPT control optimization package.

application is the common practice of sharing code among different plans. In a series of papers the researchers describe a process of separating plans using "domain-based" program understanding [15, 6, 14, 7].

There are several problems with using the domain decomposition approach for the software we have targeted. First, we are dealing with very general packages which may be used to solve a variety of problems in vastly different scientific fields. These packages may be nested within each other and are often used in very complicated ways, with multiple restarts using the same package for different purposes with each pass. It does not seem pratical to decompose plans on the basis of what type of problem is being solved. Second, these general solvers have a very large number of options which specify the numeric method to be used. The differences between methods can be very subtle and understandable only to an expert with encyclopedic knowledge of numeric methods. Decomposing domains on the basis of the method of solution would be very difficult. Third, since these are research codes there are constant changes and additions as new problems arise and new methods are developed or appear in the literature. Any sort of domain decomposition of such codes is likely to become an endless process of chasing a moving target.

Our methodology proceeds in several stages. We transform the code to a network representation with basic blocks as nodes and the direction of program execution as edges. We then color this network with three categories: decision points which are program branches controlled by user control variables, requirements blocks which are basic blocks that contain non-control user input (these may be variables set by the user or subroutines supplied by the user), and all other blocks which we label as computational blocks. The next stage is to remove all computational blocks. If we add runnable states to all the resulting leafs the result is a rough Decision Point Diagram [DPD] which is a state machine representation of all possible paths a user could take while filling in a user interface. This preliminary DPD could be transformed into a working valid interface, but it would be an annoying interface, with numerous redundancies. This preliminary DPD is then iteratively reduced using graph operations to produce a more reasonable DPD which then can be transformed into a workable interface. The result of our method is MAUI-compatible XML. This allows the quick generation of GUI skeletons as well as making XML-encoded user interface specifications readily available for other software engineering purposes.

In the next section we examine the process of producing and refining DPDs in more detail, but first we introduce our target application. DASSL: Differential Algebraic System Software [12] 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 [8]. 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 [8].

2. Methodology

Our method for deriving user interface requirements has five steps.

1. **Characterization**: Every variable and subroutine is characterized by tracing usage throughout the scope of the primary subroutine.
2. **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. **Removal of Extraneous Information**: All blocks and edges which obviously will not contribute to the solution are removed.
4. **Iterative Graph Reduction**: Graph operations are iteratively applied to the graph until no further improvement is possible.
5. **Constructing the DPD**: The reduced graph is annotated with runnable states to form a DPD.

2.1 Characterization

We begin a more formal description of our methodology with some assumptions. We assume that program values are not transferred among routines by indirection. This is a safe assumption in FORTRAN77 where indirection is possible but rarely used. It is clearly not valid for C, where language flexibility and common practice encourage wild pointer arithmetic. For the purposes of this study we deal exclusively with FORTRAN77 [F77], which is still the most commonly used programming language for this class of applications. The primary barrier to extension of this work to C is that a more complete memory emulator would be needed to trace execution through multiple levels of indirection. Similarly, we assume that applications do not use "file mediation" where data is carried from one program module to another by intermediate file storage.

We chose not to deal with these issues in part because we wish to make immediate progress in an area where there has been little traction, and in part for philosophical reasons. Our methodology essentially emulates and therefore

automates the processes which take place when a software engineer sits down to build an interface for a complex scientific application. As such it seems reasonable that in situations where a human would be confused, our methodology would also fail to make much headway.

Given the above assumptions, user input can enter a subroutine as formal parameters, a common block or as the formal parameters of a function or subroutine called from within the original subroutine. We seek to discover which of these variables, arrays or "passed-by-name" subroutine names is intended to be user input. We determine this by tracing program element usage throughout the scope of the primary subroutine. Elements which are used but never initialized (or at most initialized very early, perhaps as a default value) must be user input. We also need to determine which input variables are "control variables" which the user sets in order to determine the numerical methods employed, and therefore the general direction of the program. We define control variables to be those user inputs which appear only in program branch statements (or are initialized at most once to some default).

2.2 Network 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. We use the procedure outlined in [1] for deriving Basic Blocks, with the addition that we consider subroutine and function calls to be equivalent to jumps and returns from remote code (which in static FORTRAN they clearly are). It should be noted that Basic Block networks are directed and semi-ordered. A semi-ordered graph is one in which some siblings may be swapped with each other while others may not, depending on the type of parent node involved. If the parent is a Branch block representing an "IF", statement then the ordering of child blocks is arbitrary and therefore unordered. All other blocks are ordered among siblings from left to right. 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.

We can now define decision points, requirements blocks and computational blocks.

Definition 1 *A* decision point *is a program branch which is controlled by a user control variable.*

Definition 2 *A* requirements block *is a non-decision point Basic Block which does contain some user input.*

Definition 3 *A* computational block *is a Basic Block which does not contain any user input at all.*

After the derivation of the Basic Blocks we need to color the resulting network. Potential decision points are Basic Blocks which are branches and which contain one or more user control variables. Requirements blocks are Basic Blocks which contain any type of user input and which are not potential decision points. We now begin the process of coalescing this colored network down to a decision point diagram.

2.3 Removing Extraneous Information

All computational blocks are removed. When a block is removed, all parents of the block are connected with the removed block's child blocks. This may result in illogical situations such as a requirements block with two or more edges leading out of it. We can resolve this situation by examining the paths from the offending requirements block. If these paths share a common control variable then the corresponding decision points may be promoted until they are all children of the problematic requirements block. The sibling decision blocks are then combined, and the requirements block becomes the parent of a single decision point, solving the problem. An example of this problem and its solution is given in the case study in Section 3.

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

The resulting network is actually a DPD, which we call the "preliminary DPD". If we translated this preliminary DPD to XML and then to a GUI, we would obtain a GUI which ran the underlying application but it would be of poor quality. In the case study that follows, the preliminary DPD has numerous redundant states and is unnecessarily complex with 37 states for entering 20 variables and subroutine names. By combining states and reducing the depth and complexity of the DPD we arrive at a much more compact and logical user interface. In the case study our methodology reduces the DPD from 37 to 21 states.

2.4 Reducing the DPD Iteratively

The process of generating a compact DPD from the preliminary DPD involves moving blocks up the decision tree as far as possible. Requirements blocks which make it all the way to the DPD root combine to become default requirements. Decision points which involve the same variable and are on parallel branches may be combined and pushed up towards the parent branch. It is possible that these tasks could be accomplished in a single massive recursive traversal. For large scientific applications we may still have from several hundred to a few thousand blocks remaining in the

network at this point. From a practical standpoint, holding such a large network and all of its edges while traversing and piling iterations up on the stack will quickly exhaust the memory of even the largest machines. Instead, we have opted for an incremental approach.

The following operations are repeated until no more improvement is possible and a compact DPD has been reached.

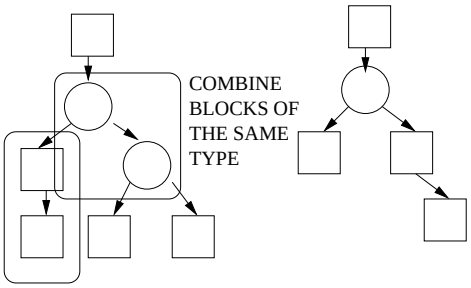

Figure 2. Combining blocks

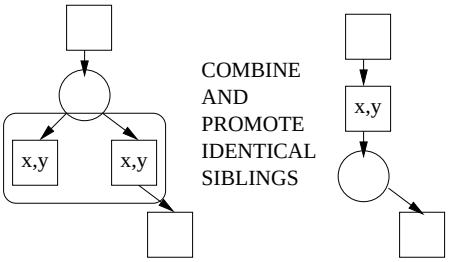

Figure 3. Combine and promote identical requirements block siblings

1. **Combine adjacent requirements blocks and adjacent decision points controlled by the same variable** (Figure 2).

2. **Swap identical requirements block siblings with their parent** (Figure 3). If all the children of a decision point are identical requirements blocks, combine them into one requirements block. This new combined requirements block is then swapped with its parent.

3. **Swap identical decision point siblings with their parent** (Figure 4). If all the children of a decision point are identical decision points, combine them into one decision point. This new combined decision point is then swapped with its parent. Non-branching series of decision points are treated as unordered so that they may be reordered to produce a situation where a combination with a decision point on another branch is possible.

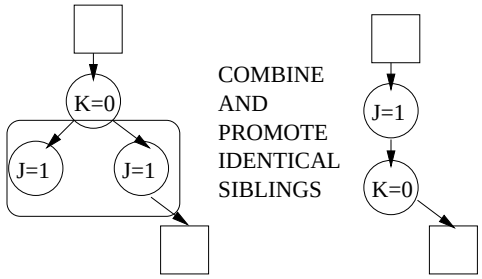

Figure 4. Combine and promote identical decision point siblings

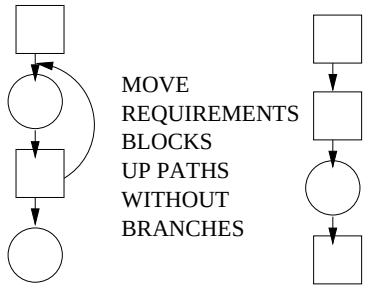

Figure 5. Move requirements blocks up non-branching paths

4. **Move requirements up non-branching paths** (Figure 5). Either of the previous two steps may result in a decision point which leads only to a single requirements block. Swap this block with its parent decision point, moving the requirements block up the decision tree. If it reaches the root then all variables and subroutines in the promoted decision block become default requirements. All other decision blocks in the DPD have these variables and subroutines removed. This may result in empty requirements blocks, which are removed.

2.5 DPD Construction

When these operations have completed the only remaining step is to add “runnable” states to the leafs. The resulting DPDs can be separated into three categories.

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 requirements.

3. **Complex Network.** DASPK is sometimes used to generate initial conditions and then restarted. This and other nesting, combination and multiple calls to the same program under differing conditions results in a complex network of decisions and requirements. In this case, determining a runnable state is more complex and may require multiple passes through our method (first for the initialization, then for the main run) in order to correctly assign runnable states.

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

3. DASPK: A Case Study

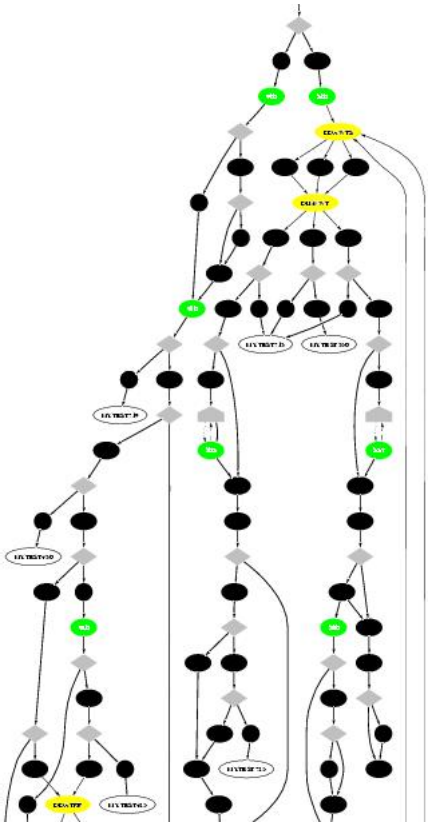

Figure 6. Network Representation.

Figure 6 shows the initial translation of a central part of DASPK to a network representation. DASPK is about 2000 lines of FORTRAN77. The full network representation has 660 blocks and 888 edges. This is much too large to display on an 8 by 11 page so we will illustrate only the derivation for the central section of DASPK, which includes the iterative-vs-direct linear solver decision. Figure 7 shows

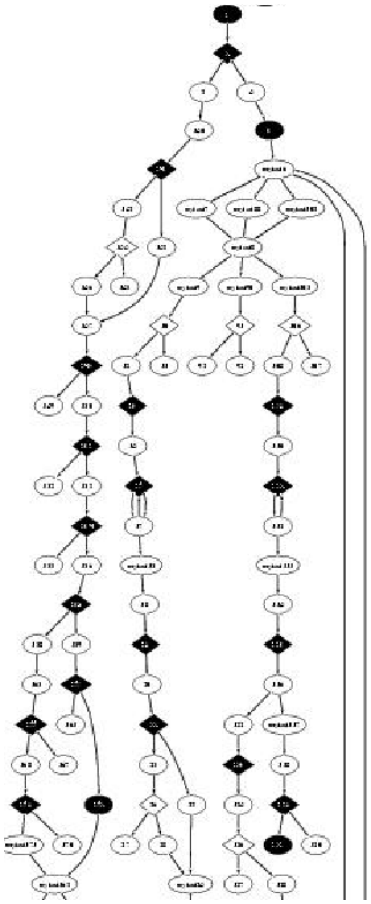

Figure 7. Network Coloring: Black diamonds represent potential decision points. Black ovals are requirements blocks. All of the other hollow blocks are computational blocks which will be removed in the next step.

the coloring of the network representation of DASPK. This central portion of DASPK has 13 control variables, 6 input variables, and 2 input subroutines.

Figure 8 shows the preliminary DPD for the central portion of DASPK after all of the extraneous blocks and edges have been removed. There are now 27 decision blocks and 10 requirements blocks.

Figure 9 shows the compact DPD after 14 iterations, at which point no further improvement is possible. The GUI corresponding to the compact DPD is close to the solution which would be arrived at by hand. The process of trans-

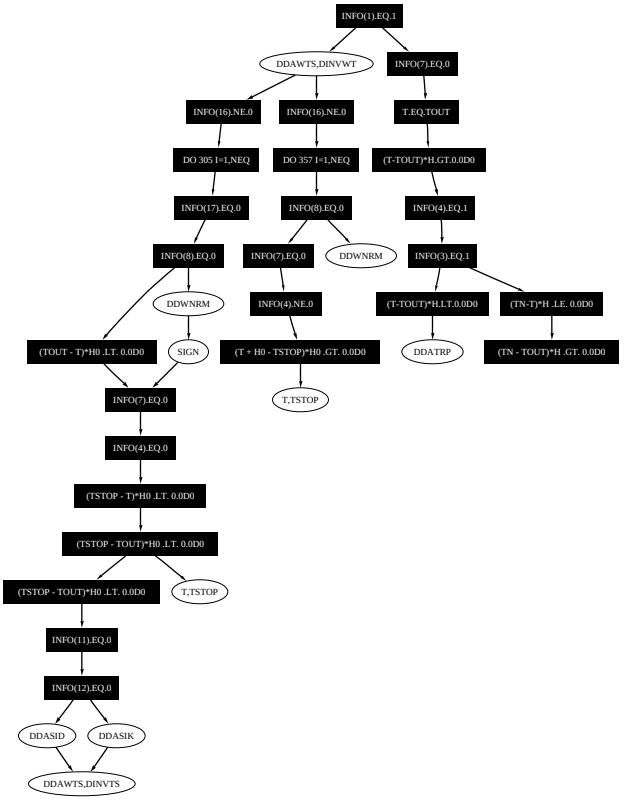

Figure 8. The preliminary DPD after all extraneous information is removed.

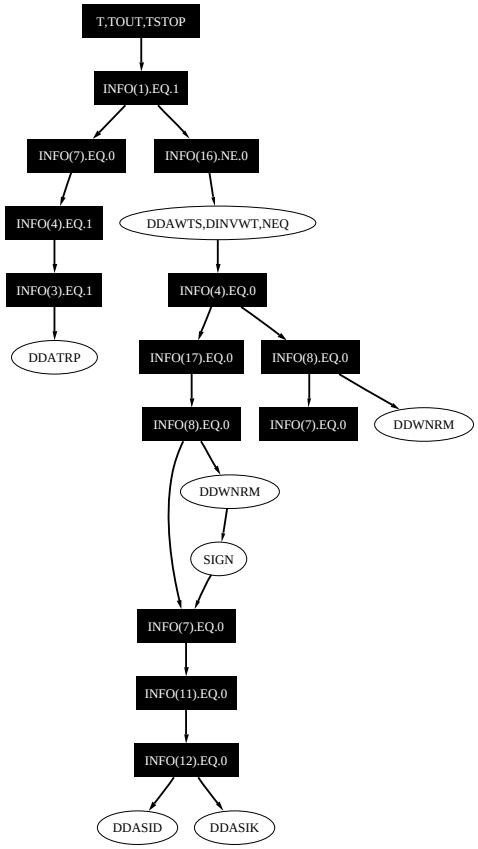

Figure 9. Compact DPD for the central section of DASPK

lating a DPD to a matching GUI is discussed in the next section.

4. Discussion

In this section we discuss two topics: the Translation of DPDs to XML to automatically produce matching GUIs, and future directions for this research.

4.1 Transforming a DPD to XML-MAUI Input

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 10. The MAUI-XML to generate the GUI in figure 10 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 sub-

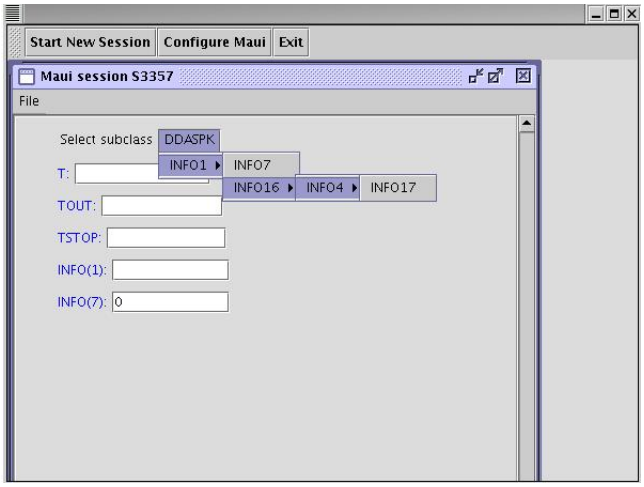

Figure 10. Maui GUI skeleton for central section of DASPK

routines can be composed, compiled and run from within the MAUI environment.

4.2 Future Directions

The iterative method of reducing DPDs produces reasonable solutions but we cannot say that they are optimal either in length of path to a runnable state or in minimizing the number of states in the DPD. In an upcoming paper we intend to introduce a new methodology which attempts to produce an optimal solution using techniques borrowed from expert systems.

The results of the MAUI XML-to-GUI technology are relatively simple GUI skeletons. Also MAUI principally targets "file input" style numeric programs which are common at the national labs. DASPK and other research codes are "compiled-in" input style which gives the user the power of an entire programming language to describe their problem. In an upcoming paper we will describe the development of an extension to MAUI which produces more complex and user friendly working environments for compiled-in research 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.Aho. *Compilers - Principles, Techniques and Tools*. Addison-Wesley, 1988.

[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] R. Clayton, S. Rugaber, and L. Wills. *Domain Based Design Documentation and Component Reuse and their Application to a System Evolution Record*. PhD thesis, Georgia Tech., October 1997. http://www.cc.gatech.edu/reverse/dare/finalreport/index.html.

[7] J.-M. DeBaudand, B. M. Moopen, and S. Rugaber. Domain analysis and reverse engineering. In *Proceedings of the 1994 International Conference on Software Maintenance*, pages 326–335, 1994.

[8] 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.

[9] M. Moore and S. Rugaber. Issues in user interface migration. In *Proceedings of the Third Software Engineering Research Forum*, 1993.

[10] M. Moore, S. Rugaber, and P. Seaver. Knowledge-based user interface migration. In *Proceedings of the 1994 International Conference on Software Maintenance*, 1994.

[11] L. Perronchon and R. Fischer. IDLE: Unified w3-access to interactive information servers. In *Proceedings of the Third International Conference on the World-Wide Web (WWW'95), Darmstadt*, 1995.

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

[13] M. Ronchetti, S. Giancarlo, and D. Feltrin. Face lift: Using www technology for an external reengineering of old applications. In *Proceedings of the Third International Conference on the World-Wide Web (WWW'95), Darmstadt*, 1995.

[14] S. Rugaber. Domain analysis and reverse engineering. White Paper, January 1994.

[15] S. Rugaber. The use of domain knowledge in program understanding. *Annals of Software Engineering*, 9, 2000. 143-192.

[16] S. Rugaber, T. Shikano, and K. Stirewalt. Adequate reverse-engineering. In *Automated Software Engineering Conference*, 2001.

[17] S. Rugaber, K. Stirewalt, and L. Wills. Detecting interleaving. In *International Conference on Software Maintenance*, 1995. 265-274.