Extending C Global Surveyor

Silvia Breu
NASA Ames Research Center/MCT
silvia.breu@gmail.com

Abstract

This paper describes an extension of C Global Surveyor (CGS), a static program analysis tool developed at the NASA Ames Research Center. CGS is used to analyse mission-critical flight software of NASA Mars missions. In order to enhance CGS and support the analysis of very large systems, the abstract interpretation techniques on which CGS is based need to be augmented by complimentary program analysis techniques. Here we describe the construction of control flow graphs that represent the programs to be analysed. This is a first step towards the application of more advanced techniques such as program slicing.

1. Motivation

Analyzing a program's structure can help to ascertain the software's strengths as well as its weaknesses. Usually, appropriate documentation is not available and programmers are not able to get the information needed to understand a system completely. Large programs thus require tools to help the developer in program understanding and debugging. The Automated Software Engineering Group at NASA Ames has developed C Global Surveyor (CGS) [1, 3], a static analysis tool based on abstract interpretation. In order to enhance CGS and support the analysis of very large systems, the abstract interpretation techniques on which CGS is based need to be augmented by complimentary program analysis techniques. We need visual tools that can help users to trace error causes through large programs. In this paper we describe the construction of control flow graphs that represent the programs to be analysed. This is a first step towards the application of more advanced techniques such as program slicing; this was the task of this internship. The next section gives a short introduction to program analysis. Section 3 describes CGS, Section 4 introduces control flow graphs and slicing, and Section 5 presents the developed extension. The last section concludes with ideas for future work.

2. Static and Dynamic Program Analysis

A wide variety of techniques has been developed which all provide additional information to the programmer. Those methods can be split into two main categories: *static program analysis* and *dynamic program analysis*.

Basically, static analysis denotes any form of analysis which does not require the execution of the system to be analyzed but surveys the source code directly. It abstracts from the code while considering all program execution paths. Static program analysis is thus *complete* and *input-insensitive*. It also is a finite approximation of a system's actual run-time behaviour. This approximation is conservative if it has no false negatives, and it is precise if it has no false positives. For example, a bug finding static analysis is conservative, if every bug is discovered—it is precise when all discovered candidates are bugs in reality. However, since the analysis takes all potential program execution paths into account, the computational effort may explode and the results may become less and less precise, if unfeasible paths are not pruned away. The best known example for static program analysis is program slicing.

In contrast to static analysis techniques, dynamic program analysis reasons about actually observed behaviour and properties of software systems. The evaluation of a program's run-time behaviour is based on a certain set of input data and thus, the information produced will be accurate but it will only hold for a particular set of program inputs. While static analysis also explores unrealizable paths, dynamic analysis even leaves feasible executions unexplored. This means that dynamic analysis is *input-sensitive* but *incomplete*. A complete dynamic analysis is not practical as it is impossible to execute all possible paths. As dynamic analysis depends on the test suites, it can detect errors but it cannot prove their nonexistence. Some established approaches are dynamic traces, delta debugging, or dynamic program slicing.

3. C Global Surveyor

C Global Surveyor (CGS) [1, 3] is a software verification tool that is used to analyse NASA software, in particular flight software systems, using static analysis techniques. It

```
set=# select * from abc_result_table;
 context | file  | function | function_id | line_number | column_number | tree_number | beg_line | exactcol_number | end_line | end_column | color
---------+-------+----------+-------------+-------------+---------------+-------------+----------+-----------------+----------+------------+-------
 ?       | set.c | main     |       65537 |          13 |            10 |          21 |       13 |               5 |       13 |          9 | G
 ?       | set.c | main     |       65537 |          14 |             8 |          33 |       14 |               3 |       14 |          7 | R
 ?       | set.c | main     |       65537 |          17 |             6 |          12 |       17 |               3 |       18 |         17 | O
 ?       | set.c | set      |       65538 |          25 |             9 |          55 |       25 |               4 |       25 |          8 | U
 ?       | set.c | set      |       65538 |          26 |            17 |           7 |       26 |              13 |       27 |         32 | O
(5 rows)
```

Figure 1. Example `abc_result_table`

focusses on the detection of run-time errors in C programs. It checks whether any operation performed in an instruction might cause a problem while the program is executed. CGS checks for problems that can occur at run-time, corrupting the memory and thus causing non-deterministic behaviour during NASA-missions. In particular, it could check for the following critical problem classes, of which only the first is implemented so far.

Out-of-bound array accesses occur when the program attempts to access an element of an a array using an index that is outside (strictly smaller or bigger than) the index bounds of the array.

Accesses to non-initialized variables occur when the program attempts to use or read a variable that has not yet been assigned a value.

De-references of null pointers occur when the program attempts to access the memory location referenced by the pointer even though the reference points to no memory location.

CGS can analyse any program written in C but the analysis algorithms have been tuned to be very precise on NASA flight software, especially from the Mars Path Finder family that includes flight software for the Deep Space 1 as well as the Mars Exploration Rover mission. The CGS implementation uses the commercial C/C++ frontend of the Edison Design Group (edg) [5]. This frontend is also part of the Green Hills' compiler [6] that is widely used at NASA for the development of flight software. It also supports a huge variety of C dialects.

The static analysis techniques used in CGS are based on *abstract interpretation*. The core idea is the formalisation of the notion of approximation: abstract interpretation, formalised by [4], is a theory of sound approximation of the semantics of computer programs, based on monotonic functions over ordered sets, especially lattices. It can be viewed as a partial execution of a computer program which gains information about its semantics without performing all the calculations: The classic example of an abstract interpretation is the sign-rule. It allows predicting the sign of an arithmetic expression before even calculating it. Assume that x is a positive number, and y is a negative number. The abstraction of the expression $x * x + y * y$ over the sign-rule allows to deduce that this expression has a positive value before calculating it.

Its main concrete application is formal static analysis, the automatic extraction of information about the possible executions of computer programs. Such analyses have two main usages, inside compilers, to analyse programs in order to decide whether certain optimisations or transformations are applicable, and for debugging or even the certification of programs against classes of bugs, as it is used in CGS. CGS's approximation is conservative in the sense that it performs all checks necessary to find all errors. In most cases, CGS can guarantee that a check is correct, incorrect, or irrelevant. The later is the case when the check refers to dead code. In some cases, the analysis cannot certify the correctness of the check, in which case it issues a warning.

CGS is designed so that it can distribute the static analysis over different processors in a cluster of machines as well as a single processor. It runs in five phases, the *initialisation-*, the *build-*, the *bootstrap-*, the *solve-*, and the *array-bound check-phase (abc)*. In the first phase, the initalisation, general information about the program (e.g., table of global variables and functions) is collected. The build computes the points-to constraints and the numerical inequalities for each program function. This is needed for the third phase: The bootstrap performs a flow-insensitive points-to-analysis and a context-independent resolution of the previously computed numerical inequalities. This is the first approximation of all memory accesses. The subsequent solve-phase performs a forward or backward interprocedural propagation of numerical invariants. The results obtained at the end are used to refine the previous results. The solve can and should be repeated until the level of precision obtained is sufficient. The last phase, the abc, checks the safety of all memory accesses based on the analysis results available, and the results (as many other results and information of the analysis) are stored in an SQL database (see Fig. 1) and flagged with the safety status of the memory access: R for certain errors, G for certain correct accesses, O for potential errors, or U for unreachable (i.e., dead code).

In summary, the CGS results *do* pinpoint the places where for example out-of-bound accesses occur, and it also stores all accesses in the SQL database, marking whether there is definitely an error, possibly an error, or no error at all. They *do not* tell what part of the code causes a particular error. Therefore, further analysis is needed to narrow down and locate the appropriate erroneous code. This is where control flow analysis and program slicing in particular can help.

4. Program Slicing and Control Flow Graphs

Program slicing is based on a very simple idea. Consider an erroneous value for a variable. Instead of searching through possibly thousands and thousands of lines of code, the programmer can compute a slice instead, that contains only the statements which may have accounted for the error. Hence, fewer lines of code have to be examined to find the bug. It is obvious that slices should be as small as possible since the needed information is retrieved much easier in slices where fewer statements are included. Therefore, an algorithm for finding the minimal slice would be worthwhile. However, this problem is undecidable [11] and every slicing algorithm can thus only provide conservative approximations of program slices.

Slicing is a functional method for automatically decomposing and reducing programs by analyzing their data and control flow. It was originally introduced by Mark Weiser in 1979 [10]. As the name implies, program slicing cuts a *slice* from the code based on a *slicing criterion*: This technique tries to calculate all the code relevant for a specific computation by throwing away the extraneous code statements. A slice is a subset of a program's statements and control predicates which (in)directly influence the values computed at the slicing criterion but do not essentially create an executable program. For a given program, there can exist many different slices for a specific slicing criterion. However, at all times, there is at least one slice for a specific criterion—the program itself.

Definition A *slicing criterion* specifies the subset of a program behaviour by a pair $\langle s, \mathcal{X} \rangle$, with s specific program statement, $\mathcal{X}$ subset of all program variables. □

(Static) slices[1] can be distinguished regarding the way they are calculated: *backward* and *forward slice*. A backward slice consists of the program statements which influence a variable's value at a specific program position, compared to a forward slice which consists of the statements being affected by modifying a variable at a particular point in the code.

To actually compute a program slice, the *control flow graph (CFG)* representation of the program can be used, as introduced by Weiser [10]. The CFG is a static representation of the program's control flow at the intraprocedural level. Each node in the graph represents a basic block, i.e., a straight-line piece of code, an executable statement, without any jumps or jump targets; jump targets start a block, and jumps end a block. Directed edges are used to represent jumps in the control flow. Edges may also be labeled with values (e.g., `true, false`) to mark under which conditions they are executed. The CFG of a program represents all alternatives of control flow, e.g., both arms of an `if`-statement are represented in the CFG. A cycle in a CFG may imply that there is a loop in the code. The CFG contains two specially designated nodes: the entry node, through which control enters into the flow graph, and the exit node, through which all control flow leaves.

```
   .
   .
   .
  <name name="A"/>
</variable_body>
<function_body id="fun4"
               storage="none"
               scope="nam0">
  <text position="fil0/3/4"/>
  <type_ref type="typ4"
            name="int"/>
  <name name="set4"/>
  <variable_body id="var1"
                 storage="auto"
                 scope="fun4">
    <text position="fil0/3/13-13"/>
    <type_ref type="typ4"
              name="int"/>
```

```
  <name name="i"/>
</variable_body>
<variable_body id="var2"
               storage="auto"
               scope="fun4">
  <text position="fil0/3/20-20"/>
  <type_ref type="typ4"
            name="int"/>
  <name name="j"/>
</variable_body>
<block>
  <text position="fil0/3/23-5/0"/>
  <return_stmt>
    <text position="fil0/4/2-12"/>
    <int_multiply type="typ4">

  .
  .
  .
```

Figure 2. **Example AST in XML representation**

Based on the CFG, slicing with respect to a certain slicing criterion is possible. This kind of analysis is also called iterative data flow analysis as the slices are computed in an iterative process: For each node in the CFG consecutive sets of relevant variables are computed. First, the directly relevant variables are determined, taking only data dependencies into account; second, variables referenced in the control predicate of `if`- and `while`-statements are determined as they are indirectly relevant if at least one of the statements in its body is relevant. This way we can calculate slices in programs, narrowing down the code that influences a certain statement in the program code.

5. Extension to CGS

The developed CGS-extension builds the intraprocedural control flow graphs of functions in C programs and stores it in the database used by CGS. We focussed on intraprocedural control flow graphs, as we mostly want to analyse specific functions in the C programs and not the flight software system as a whole. Moreover, we first assume that the reason for array-out-of-bounds errors is mostly located in the same function.

As CGS relies on the edg-frontend, we use the same frontend to generate the abstract syntax tree (AST) of the analysed program as representation in the Extensible Markup Language (XML) [12] (see Fig. 2). The XML-file is parsed in order to retrieve the AST in internal Java object representation. Therefore, we used *ccast* [2], a tool that was developed in the ASE group. ccast offers all necessary interfaces, and (abstract) classes to represent abstract syntax trees for both C and C++ programs as objects, e.g., `Variable, VariableBody, FunctionBody, BinaryOperator, AssignRightShift, WhileStmt, CaseStmt` and many more.

1 Slices can be differentiated by static and dynamic slices: Static slices are computed without making assumptions in terms of any program input whereas the dynamic variant relies upon some specific test cases.

```
 1 int A[10];
 2
 3 int set(int i, int j) {
 4   return i*j;
 5 }
 6
 7 main () {
 8   int *p, j, i;
 9   int k;
10
11   char c;
12
13   p = &A[1];
14   j = 5; k = 7;
15
16   for (i=1; i<7; i++) {
17     while (j<10) {
18       do {
19         if (i < 10) {
20           i = i + set(i,i);
21           continue;
22           i++;
23         } else {
24           break;
25           p[i] = set(3,19);
26         }
27       } while (j+i<15);
28       j=17;
29       break;
30     }
31     continue;
32   }
33
34   switch(c){
35     case 'Y':
36       c='y';
37       break;
38     case 'N':
39       c='n';
40       break;
41     default:
42       c=' ';
43   }
44
45   goto NoHands;
46
47   for(k=10; k>1; k--) {
48   burp: --k;
49   }
50
51   if(j < 100) {
52     goto burp;
53   }
54
55  NoHands:
56   k=42;
57
58   switch(c) {}
59
60   switch(c) {
61   case 'A'+1:
62   case 3:
63     ;
64   }
65 }
66
67 int add(int i, int j) {
68   return i+j;
69 }
```

Figure 3. **Example C code** `weirdo.c`

The actual computation of the control flow graphs is performed by traversing the AST. The tool takes as parameters C-files and function names; thus the user can define what control flow graphs should be built (for our running example `weirdo.c` in Fig. 3 we specified only `set` and `main` but not `add`). Instead of a huge distinction over more than 400 different object types that can exist in an AST for a C/C++ program, a visitor following the visitor pattern is used to traverse the AST correctly. The visitor implements the necessary actions for building the CFG when visiting the different nodes of the AST. To store the calculated control flow graphs, a representation in the relation database, also used by CGS, was chosen: This way, it is possible to easily extend the analysis in any programming language, and to use the CFG for further analysis, provided that an interface to access a SQL database exists. As we already had to use the edg-frontend, an XML parser, ccast implemented in Java, CGS written in C with the results stored in a PostgreSQL relational database, there were enough limitations and problems to overcome. To make things easier for any user of the built control flow graphs, we decided to store the CFG in the same database CGS uses.

The database table `stmt_table` stores all nodes (see Fig. 4), with the information about file- and function-name, line- and column-number of the beginning of the statement represented by the node, as well as the type of the node, and the information whether the statement contains any calls to other functions. For example, line *20* in Fig. 3, represented by node with `stmt_id` 16 in Fig. 4 contains a function call, which is indicated with the boolean flag `true` for the attribute `with_call` in the database table. We can see that in the case of `switch`-statements, the edg-frontend unfortunately does not provide the exact location, where the different `case`-statements begin (e.g., see nodes 24, 25, 27, 28, 30, 31, 33 in Fig. 4 for lines *34–43* in Fig. 3). Thus,

```
CGS=# select * from stmt_table;
 stmt_id |   file   | function | line_number | column_number |     type     | with_call
---------+----------+----------+-------------+---------------+--------------+-----------
       0 | weirdo.c | set      |          -1 |            -1 | ENTER        | f
       1 | weirdo.c | set      |           4 |             2 | ReturnStmt   | f
       2 | weirdo.c | set      |          -1 |            -1 | EXIT         | f
       3 | weirdo.c | main     |          -1 |            -1 | ENTER        | f
       4 | weirdo.c | main     |           8 |             7 | VariableBody | f
       5 | weirdo.c | main     |           8 |            10 | VariableBody | f
       6 | weirdo.c | main     |           8 |            13 | VariableBody | f
       7 | weirdo.c | main     |           9 |             6 | VariableBody | f
       8 | weirdo.c | main     |          11 |             7 | VariableBody | f
       9 | weirdo.c | main     |          13 |             2 | EvalStmt     | f
      10 | weirdo.c | main     |          14 |             2 | EvalStmt     | f
      11 | weirdo.c | main     |          14 |             9 | EvalStmt     | f
      16 | weirdo.c | main     |          20 |             3 | EvalStmt     | t
      17 | weirdo.c | main     |          21 |             3 | ContinueStmt | f
      18 | weirdo.c | main     |          22 |             3 | EvalStmt     | f
      19 | weirdo.c | main     |          24 |             3 | BreakStmt    | f
      20 | weirdo.c | main     |          25 |             3 | EvalStmt     | t
      15 | weirdo.c | main     |          19 |             1 | IfStmt       | f
      14 | weirdo.c | main     |          18 |             6 | DoStmt       | f
      21 | weirdo.c | main     |          28 |             6 | EvalStmt     | f
      22 | weirdo.c | main     |          29 |             6 | BreakStmt    | f
      13 | weirdo.c | main     |          17 |             4 | WhileStmt    | f
      23 | weirdo.c | main     |          31 |             4 | ContinueStmt | f
      12 | weirdo.c | main     |          16 |             2 | ForStmt      | f
      25 | weirdo.c | main     |          34 |             2 | CaseStmt     | f
      26 | weirdo.c | main     |          36 |             6 | EvalStmt     | f
      27 | weirdo.c | main     |          34 |             2 | BreakStmt    | f
      28 | weirdo.c | main     |          34 |             2 | CaseStmt     | f
      29 | weirdo.c | main     |          39 |             6 | EvalStmt     | f
      30 | weirdo.c | main     |          34 |             2 | BreakStmt    | f
      31 | weirdo.c | main     |          34 |             2 | DefaultStmt  | f
      32 | weirdo.c | main     |          42 |             6 | EvalStmt     | f
      33 | weirdo.c | main     |          34 |             2 | BreakStmt    | f
      24 | weirdo.c | main     |          34 |             2 | SwitchStmt   | f
      34 | weirdo.c | main     |          45 |             2 | GotoStmt     | f
      36 | weirdo.c | main     |          48 |             2 | LabelStmt    | f
      37 | weirdo.c | main     |          48 |             8 | EvalStmt     | f
      35 | weirdo.c | main     |          47 |             2 | ForStmt      | f
      39 | weirdo.c | main     |          52 |             4 | GotoStmt     | f
      38 | weirdo.c | main     |          51 |             2 | IfStmt       | f
      40 | weirdo.c | main     |          55 |             1 | LabelStmt    | f
      41 | weirdo.c | main     |          56 |             2 | EvalStmt     | f
      42 | weirdo.c | main     |          58 |             2 | SwitchStmt   | f
      44 | weirdo.c | main     |          60 |             2 | CaseStmt     | f
      45 | weirdo.c | main     |          60 |             2 | CaseStmt     | f
      46 | weirdo.c | main     |          60 |             2 | BreakStmt    | f
      43 | weirdo.c | main     |          60 |             2 | SwitchStmt   | f
      47 | weirdo.c | main     |           7 |             8 | ReturnStmt   | f
      48 | weirdo.c | main     |          -1 |            -1 | EXIT         | f
(49 rows)
```

Figure 4. **CFG nodes for** `set` **and** `main`

we decided to choose the beginning of the corresponding `switch`-statement. Additionally, the table also contains the dedicated entry and exit nodes of each computed control flow graph which are necessary as described in Section 4. They have type `ENTER` and `EXIT` resp. (see Fig. 4).

`stmt_succ_table` represents the edges (see Fig. 5), determined by the numbers of source and target CFG node, in-

```
CGS=# select * from stmt_succ_table;
 src | target |  label  | jump
-----+--------+---------+------
   0 |      1 |         | f
   1 |      2 |         | f
   3 |      4 |         | f
   4 |      5 |         | f
   5 |      6 |         | f
   6 |      7 |         | f
   7 |      8 |         | f
   8 |      9 |         | f
   9 |     10 |         | f
  10 |     11 |         | f
  11 |     12 |         | f
  12 |     13 | true    | f
  13 |     15 | true    | f
  15 |     16 | true    | f
  16 |     17 |         | f
  17 |     14 |         | t
  15 |     19 | false   | f
  14 |     15 | true    | f
  20 |     14 |         | f
  18 |     14 |         | f
  14 |     21 | false   | f
  19 |     21 |         | t
  21 |     22 |         | f
  13 |     23 | false   | f
  22 |     23 |         | t
  23 |     12 |         | t
  12 |     24 | false   | f
  24 |     25 | 89      | f
  25 |     26 |         | f
  26 |     27 |         | f
  24 |     28 | 78      | f
  28 |     29 |         | f
  29 |     30 |         | f
  24 |     31 | default | f
  31 |     32 |         | f
  32 |     33 |         | f
  27 |     34 |         | t
  30 |     34 |         | t
  33 |     34 |         | t
  35 |     36 | true    | f
  36 |     37 |         | f
  37 |     35 |         | f
  35 |     38 | false   | f
  38 |     39 | true    | f
  38 |     40 | false   | f
  40 |     41 |         | f
  41 |     42 |         | f
  42 |     43 | default | f
  43 |     44 | 3       | f
  44 |     45 |         | f
  43 |     45 | 66      | f
  45 |     46 |         | f
  43 |     47 | default | f
  46 |     47 |         | t
  39 |     36 |         | t
  34 |     40 |         | t
  47 |     48 |         | f
(57 rows)
```

Figure 5. **CFG edges for** `set` **and** `main`

cluding the information whether the edge represents a jump, and whether the edge is labeled and if labeled with what value. Here, due to the edg-frontent, we have the limitation that we cannot retrieve the information whether the label is an integer value or the ASCII value of a character (see `switch`-statement in lines *60–64* in Fig. 3 and the appropriate labels for edges $43 \longrightarrow 44$, and $43 \longrightarrow 45$ resp., labeled with 3 and 66 resp. in Fig. 5).

For convenience, a database table `goto_table` separately encapsulates all jumps in the control flow, including the information whether the jump is forward or backwards in the code, and whether the jump stays in the scope of the current control structure or not.

```
CGS=# select * from goto_table;
 src | target | forward | in_scope
-----+--------+---------+----------
  39 |     36 | f       | f
  34 |     40 | t       | t
(2 rows)
```

Another convenient table is `cfg_list_table` which stores all already computed control flow graphs, specified by file- and function-name.

```
CGS=# select * from cfg_list_table;
   file   | function
----------+----------
 weirdo.c | set
 weirdo.c | main
(2 rows)
```

6. Future Work

The extension of CGS by the CFG-construction enables the integration of further program analysis techniques. There exist several directions in which future work could continue:

- extension of CFG for C++
- slicing using CFG
- computation of program dependence graph (PDG)
- slicing using PDG
- extension for multi-threaded software

In order to also support C++ programs, e.g., software developed at the Jet Propulsion Lab (JPL), additional nodes in the AST like exceptions, or templates have to be treated. For that, ccast already offers the necessary classes for the internal object representation of the AST. On top of the CFG we could implement Weister-style slicing.

Building the appropriate *program dependence graph* offers even more and better possibilities to analyse the programs. The PDG eases the task of slicing: The computation of the slice, based on a certain slicing criterion which can be extraced easily from the `abc_result_table` of CGS, is then reduced to a simple graph reachability problem. PDG-based slicing is a very common technique.[2]

A PDG is a directed graph $G = \langle \mathcal{V}, \mathcal{E} \rangle$ with $\mathcal{V}$ set of vertices denoting program statements and control predicates, and $\mathcal{E}$ set of edges representing data and control dependences between statements. Data dependence of two statements means that the value of a variable set in one statement is used by means of evaluating that variable at another statement, whereas control dependence between two statements exists if one statement controls whether and how often the other statement is executed.

Having calculated the PDG of a certain program, the slicing criterion is now a single vertex in the graph as each $v \in \mathcal{V}$ does not only denote a specific program statement but also the program variable being assigned by that particular statement. Thus, determining forward- and backward-slice $S^{+}(s)$ and $S^{-}(s)$ based on a criterion s is quite simple: $S^{+}(s) = \{v \in \mathcal{V} \mid \exists s \rightarrow^{*} v\}$ (all vertices which are reachable from the vertex representing the slicing criterion), while $S^{-}(s) = \{v \in \mathcal{V} \mid \exists v \rightarrow^{*} s\}$ (all vertices from which the considered vertex is reachable). Note, that $u \rightarrow^{*} v$ is the reflexive transitive closure of $(u, v) \in \mathcal{E}$ with $u, v \in \mathcal{V}$.

Furthermore, there already exist approaches to slice multithreaded programs based on control flow and program dependence graphs, e.g., [7].

References

[1] Guillaume Brat and Arnaud Venet. *Precise and Efficient Static Array Bound Checking for Large Embedded C Programs.* In Proc. of PLDI 2004, pp. 231–242, Washington, DC, June 2004. ACM.

[2] Owen O'Malley et. al. *Propel: Tools and Methods for Practical Software Model Checking.* Workshop on Model Checking for Software Intensive Systems at Dependable Systems and Networks Conference, June 2003.

[3] C Global Surveyor. `http://ase.arc.nasa.gov/brat/cgs/`.

[4] Patrick Cousot and Radhia Cousot. *Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixedpoints.* In Proc. of 4th POPL, pp. 238–252, London-New York, Jan. 1977. ACM.

[5] Edison Design Group. `http://www.edg.com/`.

[6] Green Hills Software. `http://www.ghs.com/`.

[7] Jens Krinke. Static Slicing of Threaded Programs. In Proc. of PASTE, 33(7):35–42, Montreal, June 1998.

[8] The PostgreSQL Global Development Group. `http://www.postgresql.org/`.

[9] Frank Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3(3):121–189, Sep. 1995.

[10] Mark Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method.* PhD thesis, University of Michigan, 1979.

[11] Mark Weiser. *Program Slicing.* IEEE Transactions on Software Engineering, 10(4):352–357, July 1984.

[12] Extensible Markup Language. `http://www.xml.com/`.

2 A good overview of different slicing techniques as well as links to continuative and more detailed literature about slicing can be found in [9].