RC 21484 (96946) 25MAY99 Computer Science

Research Report

Error Detection by Data Flow Analysis Restricted to Executable Paths

D. Brand
IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center [Publications 16-220 ykt] P. O. Box 218, Yorktown Heights, NY 10598. email: `reports@us.ibm.com`
Some reports are available on the internet at `http://domino.watson.ibm.com/library/CyberDig.nsf/home`

Error Detection by Data Flow Analysis Restricted to Executable Paths

D. Brand
IBM T. J. Watson Research Center, Yorktown Heights, NY, USA

May 25, 1999

Abstract

BEAM is a tool for finding errors in C++ programs by source code analysis. It uses data flow analysis to find paths leading to an error. Classical data flow analysis propagates information along all paths, including non-executable ones, which results in reporting errors that are not real. To avoid this problem, BEAM restricts data flow analysis to paths that are executable, and in addition also consistent with user's assumptions on initial state of memory. However, the user is not required to specify his assumptions. The main contribution of this paper is an algorithm for finding an executable path to error, while avoiding path enumeration.

1 Introduction

The objective of BEAM (Bugs Errors And Mistakes) is to reduce software development time by identifying errors early. An example of an error BEAM looks for is shown in Figure 1(a), where the procedure `P` may return an uninitialized value of `X`.

There are many approaches to this problem, and they differ in their trade-offs between run time, maximizing the number of errors found, and avoidance of "bogus" errors (i.e. what the user does not consider a real error). For compilers and lint programs speed is a high priority, which does not allow time to decide

```
int P ( int A)
{
  int X;
  int I = 0;

  if ( A ) { X = 0; I = 1;}

  if (!I ) return X;
  else     return 0;
}
```

```
int P ( int A)
{
  int X;
  int I = 0;

  if ( A ) { X = 0; I = 1;}

  if ( I ) return X;
  else     return 0;
}
```

Figure 1: (a) Uninitialized X returned (b) X is initialized when returned

whether a potential error can really happen or not. Given this dilemma, compilers prefer to report only what is very likely a to be real error, while lint programs prefer to maximize the number of errors reported, even if many of them are not real. For verifiers, detection of all errors is the highest priority, but reporting of bogus errors is acceptable; that is, specification that is not sufficient to prove the absence of error is considered an error in itself.

BEAM shares some goals with compilers, in that it wants to avoid bogus errors without requiring any specification from the user. (While no specification of individual procedures is required, project specific information is commonly provided, and used during analysis.) In contrast with compilers, we allow longer run times, and like a verifier, BEAM spends the time on ensuring that paths leading to an error are actually executable. For example, we want to detect the error in Figure 1(a) without issuing any message about any potentially uninitialized variable in Figure 1(b). (The variable `X` would be uninitialized in Figure 1b along the path corresponding to `A == 0` and then `I != 0`. But that path is not executable.)

To find errors in a given procedure, while avoiding bogus errors, four problems must be solved.
1) We need to discover all side-effects of called procedures.
2) We need to discover what assumptions the user is making about initial state of memory.
3) We need to be able to determine whether a given path is executable.
4) We need to find a path leading to error, which is executable under the assumptions on initial state of memory.
While problems 1) and 3) are very important, neither inter-procedural analysis nor theorem proving are subjects of this paper; solutions to these two problems will be assumed, but not described. Only problems 2) and 4) are addressed. The challenge for problem 2) is to discover user's assumptions without requiring the user to specify them, because that is considered too much of a burden. The challenge for problem 4) is to find an executable path without enumeration of all paths, which would be infeasible.

Problems 1) and 3) are unsolvable in general, and problem 2) is not even well defined because it involves guessing user's intentions. Therefore we cannot expect exact solutions. However, suppose there were oracles for solving problems 1), 2) and 3). Solutions to problems 1) and 2) would require some higher order language, but assume that the oracle for problem 3) is able to decide satisfiability of any set of predicates in that higher order language. Then clearly problem 4) could be solved exactly; however to guarantee that an executable path is found for every error, an exponential search for such a path might be needed. The contribution of this paper lies in a method that reduces the need for path enumeration.

The input to BEAM consists of the same source files as needed for compilation. The output is a list of errors, of several kinds. In this paper we will consider only one kind, namely errors expressible as reaching a particular statement. This includes failing assertions, dereferencing NULL pointers, indices out of range, etc. This excludes uninitialized variables, memory leaks, and others which require a different kind of data flow analysis. This also excludes stylistic errors and other types of errors, which do not require data flow analysis at all. (Section 6 will include data on all these types of errors including those which are not discussed.) For each error considered in this paper the user is also given a path, which is executable and which terminates in the error.

The main method of BEAM is data flow analysis, which is commonly used to detect errors [1]. Traditional data flow analysis [2] propagates information along all paths, including those that are not executable; such analysis cannot distinguish between the programs of Figure 1(a) and 1(b). Conditional constant propagation [3] is an improvement in that it avoids dead code in its analysis. In [4], slicing [5, 6] is restricted to paths satisfying given input constraints. An approach restricting slicing to executable paths is presented in [7]. Our approach can be considered an extension of [8], which performs data flow analysis avoiding some unexecutable paths. Although their algorithm could not distinguish the programs of Figure 1(a) and 1(b), it

```
int P ( int A)
{
  int X;
  int I = 0;

  if ( A ) { X = 0; I = 1;}

  if (!A ) return X;
  else     return 0;
}
```

```
int P ( int A)
{
  int X;
  int I = 0;

  if ( A ) { X = 0; I = 1;}

  if ( A ) return X;
  else     return 0;
}
```

Figure 2: (a) Uninitialized X returned (b) X is initialized when returned

could distinguish the programs of Figure 2(a) and 2(b).

There are two major differences between unrestricted data flow analysis, and analysis restricted to executable paths. The first is efficiency. Executability is undecidable in general, and even if we sacrifice completeness, we can expect to spend significant amounts of time. Moreover, memory requirements are larger because nothing in a given program can be abstracted away; every detail could make a difference between a path being executable or not. As a result, we cannot expect to run at compiler speed; BEAM's goal is to run overnight on programs modified during the day.

The second issue is convergence in the presence of loops and recursive procedures. Classical data flow analysis and model checking [9, 10, 11] (which are closely related [12, 13]) achieve convergence by abstracting data values, so that data domains contain only a finite number of elements. For example, many data flow analyses consider merely whether a variable is assigned, while ignoring the actual value assigned; or instead of considering all possible integer values, only values modulo a constant may be considered; or all possible states of a queue may be abstracted into the three states {empty, full, partially filled}. Such abstractions are safe provided any property derived about the abstracted model is also true about the original one. Safety is assured for properties containing universal quantification only, such as correctness properties ("For all input values no error will occur"). But safety is not assured for properties containing existential quantification ("There exist input values for which an error will occur"). As a result, abstracting the data domain may lead to reporting errors which cannot occur in the original program. Since our goal is to avoid exactly this problem (at the expense of failing to detect some errors), we cannot use any data abstraction. Therefore to achieve convergence we do control flow abstractions only. The potentially unbounded execution of a recursive procedure is represented, as usual, by a single call statement.

While inter-procedural analysis is not the subject of this paper we need to clarify a few points. In general, procedure calls are handled in two ways. Some procedures are expanded inline; the criteria for deciding inlining are rather heuristic and are not discussed. Side-effects of all the remaining procedure calls are represented by procedure summaries, calculated in a separate step.

Loops are represented as recursive procedures and treated as such. In our implementation each loop is unrolled once and the remaining iterations of the loop are represented by a call to the loop-representing procedure. The information about the remaining iterations is expressed in the procedure summary describing, for example, which variables are assigned, but we do not know the exact conditions under which the assignments might happen. As a result, errors that can occur only if a loop is executed more than twice will

```
int P ( int *A, int *B )
{
  if (!A)
    printf("internal error\n");

  if (!B)
    abort();


  return *A + *B;
}
```

Figure 3: Examples of error statements

not be reported. However, this abstraction is clearly safe in the sense that for each error we can make sure that there is an executable path leading to it. (Treating loops as recursive procedures implies that we cannot handle backward gotos.)

The paper is organized as follows. Section 2 will discuss problem 2) above (i.e., guessing what inputs should be considered legal). Section 3 will defines the graph on which data flow analysis is performed and how a solution to problem 2) is represented. Section 4 describes those aspects of problem 3) (i.e. theorem proving) that are needed by the algorithms in Section 5.

2 Conditions for reporting an error

The goal of this section is to discover what a user would consider a "real" error. In particular, it discusses how to guess which combinations of inputs are considered legal. One could take the attitude that legal inputs of a given procedure can be determined from all the calls to that procedure. That approach is not applicable for two reasons. First we may not be given the whole system from which the procedure could be called; this is particularly true about general purpose libraries. Secondly, even if we were given the whole system, we could not account for future modifications. Therefore we need to guess what the user might have written in an informal description of his procedure.

Solutions to this problem depend to a large degree on the personality of individual users. Therefore the purpose of this section is not to provide a definitive solution, but merely to motivate the representation of possible solutions, because that representation will be used in the rest of the paper.

2.1 Statements considered an error

We assume that any statement can be marked as "error"; for example, in Figure 3 execution of the print statement or the abort statement would be typically considered an "error". The policies for marking statements as error are mentioned in Section 2.4.

When a statement is declared to be an error, it has two implications. First BEAM tries to find conditions that will cause the error statement to be reached. Secondly, once an error statement is reached, no later errors should be reported. For example, in Figure 3 we do not want to report about the possibility of `A` or `B`

```
int P ( int *X )
{
  return *X;
}
```

Figure 4: There is no evidence that X will ever be NULL

```
int P ( int *X )
{
  if (X)
    printf("All is well.\n");

  return *X;
}
```

Figure 5: There is evidence that X may be NULL.

being `NULL` when dereferenced, because such a situation would be necessarily preceeded by another error statement.

Some statements will result in an error only conditionally; for example, `return *X;` would fail only if X were NULL. For the purposes of exposition we will assume that such conditional errors have been rewritten to be unconditional (although BEAM performs these replacements only implicitly). For example, `return *X;` would be replaced by `if (!X) abort(); else return *X;`,

2.2 Multi- and single-statement criterion

In Figure 4 consider the question of whether the user should be told that the dereferencing of `X` will be an error if `X` were `NULL`.

One approach [14] is to require the user to declare which parameters may be `NULL` and which not; that allows checking both inside procedures and across procedure calls. That approach is not acceptable in our environment, because it would require modifying existing programs. Instead of placing the burden of proof on the user to show why an error cannot occur, we place the burden of proof on the tool to show that an error can occur. Since Figure 4 contains no evidence one way or the other, BEAM will not issue any error message.

BEAM takes the same approach as [15], which will flag the dereferencing of a pointer as an error only if there were evidence that the user expected the possibility of the pointer being `NULL`. Thus no error would be flagged in case of Figure 4, but the program in Figure 5 would be considered wrong because the test of `X` is evidence that the author had thought of the possibility that `X` might be `NULL`. We would consider Figure 4 to contain an error only if during the analysis of some other procedure sufficient evidence was found that `P` could be called with a `NULL` argument.

This approach is not perfect for two reasons. First it misses many real errors, but more seriously it also produces bogus error reports. The reason is that some users feel that the test of `X` in Figure 5 is not sufficient

```
int P ( int A, int B )
{
  int *X = NULL;

  if ( A ) X = &M;
  if ( B ) X = &N;

  return *X;
}
```

Figure 6: Is there enough evidence that X may remain NULL?

```
int P ( int A, int B )
{
  assert ( A || B);

  int *X = NULL;

  if ( A ) X = &M;
  if ( B ) X = &N;

  return *X;
}
```

Figure 7: No error reported even with the multi-statement criterion.

evidence that `X` could be `NULL`. However, everybody agrees that there is an inconsistency in the treatment of `X`.

Now consider the example in Figure 6. Should the user be told that the dereferencing of `X` will be an error if both `A` and `B` are 0? The two tests provide evidence that both `A` and `B` could be 0, but there is no evidence that they could be 0 at the same time. Some people do not want to see any error reported in this case, while others do, because they want to be forced to use an assert statement as in Figure 7. BEAM supports both criteria for reporting an error and refers to them as “single-statement criterion” and “multi-statement criterion”.

To understand the two criteria, consider Figure 5. The if-statement does not have any else clause, but we can assume an else clause with the empty statement. If that else clause were ever executed that would necessarily imply the dereferencing of a `NULL` pointer. We take the position that every statement (including omitted else clauses) is meant to be reachable; thus we can use its reachability condition to try to prove conditions for an error.

In Figure 6 the reachability of the two unspecified else clauses implies dereferencing of a `NULL` pointer. However, there is no single statement whose reachability implies that `X` will remain `NULL`. The two criteria will be defined exactly in Section 5; informally, the multi-statement criterion considers a condition for an error to be satisfied if the condition is implied by the reachability of some number of statements. The stricter single-statement criterion requires a single statement whose reachability implies that an error will occur.

2.3 Admissible and inadmissible conditions

In Section 2.2 we said that we assume that the user meant every statement to be reachable. That is not true in general, with error statements being a typical example. For example, in Figure 3 it would not be useful to tell the user that the return statement will fail if `A` were `NULL`; he is clearly aware of that. If the user were to provide specifications of legal procedure inputs he would most likely consider NULL to be illegal. This is in contrast to Figure 5 where he would consider NULL to be a legal input and therefore the return statement of Figure 5 should be considered in error.

As mentioned earlier we use conditionals in the program as an indication of what input combinations the user considers legal. But as Figure 3 shows not all conditions can be used as evidence of legal input; we call such conditions "inadmissible", while conditions such as those in Figure 5 are call "admissible". Only admissible conditions may be used as evidence that the user expected the possibility that the tested condition will be sometimes true and sometimes false.

It is not for the user to specify which conditions are admissible and which are not; that is determined by a particular project's policy (Section 2.4) and also by heuristics. Any condition that leads directly to an error is usually considered inadmissible, while other explicitly appearing conditions are considered admissible. For example, when given

```
if (A)
  if (B)
    abort();
```

the normal BEAM heuristic would declare the test of `A` as admissible, while the test of `B` would be inadmissible.

The choice of which conditions should be considered admissible has a great impact on the number of errors reported. The more conditions are considered admissible, the more errors will be reported.

2.4 Policy

Before analysis can start three kinds of decisions need to be made.
I) Which statements constitute an error?
II) For each error statement should evidence against it satisfy the single- or multi-statement criterion?
III) Which conditions are admissible and which are inadmissible?

Question I) is fairly straightforward. There are three kinds of error statements – those determined by the semantics of the programming language (e.g., abort, dereferencing a null pointer, index out of range), those determined by heuristics (e.g., any print statement containing the word "error" is an error statement), and those determined by a particular project (e.g., BEAM may be told that any call to the procedure `InternalError()` should be considered an error).

For questions II) and III) we consider three policies.

The strictest policy: Only the multi-statement criterion is used. All conditions are admissible, except those declared to be procedure preconditions. This is the policy used in verification environments, where the goal is to find all errors. In our environment this policy is not used because it would require specifying all procedure preconditions.

The medium policy: The multi-statement criterion is used on all error statements, except asserts, for which the single-statement criterion is used. All conditions are admissible, except those leading immediately to an error statement, or those introduced not by the user, but by BEAM.

The loosest policy: Only the single-statement criterion is used. Inadmissible conditions are those of the medium policy plus all conditions inside loops and inside invoked procedures.

```
int P ( int N )
{
  int *X = NULL;
  int I;

  for (I = 0; I < N; I++)
    if (F(I))
      X = &N;

  return *X;
}
```

Figure 8: Dereferencing of X is an error if F(I) is false for all I.

```
int P ( int N )
{
  int *X = NULL;
  int I;

  for (I = 0; I < N; I++)
    if (F(I))
      X = &N;

  assert( X );

  return *X;
}
```

Figure 9: The assert will fail if N is not positive.

We will illustrate the difference between the medium and the loosest policy by the example of Figure 8. The variable `X` will remain `NULL` if none of the loop iterations can satisfy the condition `F(I)`. With the medium policy, BEAM will report that `X` may be `NULL` when dereferenced; this happens, for example, at the end of the path which goes around the loop exactly once without satisfying the condition `F(0)`. The user may not consider this to be an error, if before calling `P`, global data structures are supposed to be arranged so that there always is an index `I` satisfying `F(I)`. The easiest way of declaring this property is to use the assert

```
int P ( int N )
{
  assert( N > 0 );

  int *X = NULL;
  int I;

  for (I = 0; I < N; I++)
    if (F(I))
      X = &M;

  assert( X );

  return *X;
}
```

Figure 10: No error reported.

statement as in Figure 9. This will stop BEAM from complaining about dereferencing of `X`, and concentrate on finding ways to violate the assertion. But the assertion can be violated in exactly the same way, which is not what the user wants to hear. Therefore it is important that for assertions we use the single-statement criterion, which would not accept the above path going around the loop exactly once. If we had used the loosest policy then there would had been no need to add the assertion because the single-statement criterion is applied to all error statements.

With the medium policy BEAM will still find that the assert statement in Figure 9 will fail, namely in case the loop is never executed. The test `I < N` on the very first iteration, when `I = 0`, is used as evidence of the possibility that `N <= 0`. If the user is sure that `P` will never be called with such a value of `N`, he needs to add the assert statement as in Figure 10. In contrast, with the loosest policy there is no need for such an assertion because the test `I < N` is considered inadmissible, and thus cannot be used as evidence for the possible values of `N`.

In the following sections we will ignore the questions of policy and assume that the decisions I), II), III) have been made in some arbitrary way.

3 Representation

BEAM does not analyze the whole program at once, but only one procedure at a time, including procedures called within. In principle analyzing the main procedure would then include the whole program, but the usual limitations of inter-procedural analysis imply that we cannot expect to catch all error just by analyzing main.

From now on we assume a fixed procedure to be analyzed. The procedure is represented by a graph containing representation of all the data operations as well as the control flow. As a result of our handling of loops and recursive procedures (see Section 1), the graph is acyclic. Figure 11 is an example of a control graph; it is the graph for the program of Figure 6. Nodes and edges in the control graph rely on results

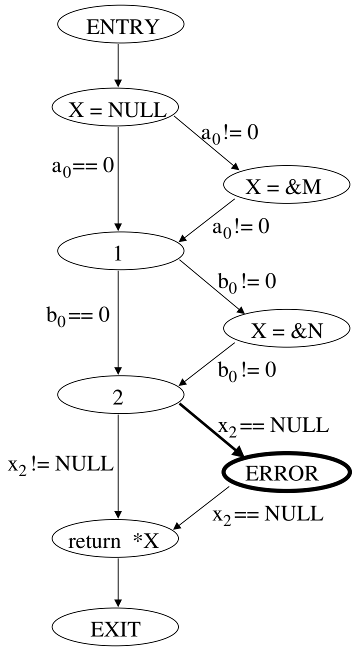

Figure 11: Graph of Figure 6

of data operations; in particular edges are labeled with conditions for execution, which are conditions on results of some data operations.

We will deal only with the control graph, which will be defined below, and here we state only briefly some properties of the data operations. Data operations include the usual arithmetic, relational, and other operators. One important operation is fetching data from memory. A fetch node takes as input a memory location and a node in the control graph; its output represents the contents of the memory location after executing the node in the control graph.

There is a representation of all the parameters of the procedure being analyzed as well as a representation of the initial state of global memory; these are called **independent parameters**. If given specific values for all the independent parameters, it is possible to evaluate all the data operations to obtain values of **dependent parameters**, which are inputs into the edge conditions. For example, in Figure 11 the dependent parameters are a_0, b_0, x_2. The value of a_0 is the result of fetching from memory the contents of the variable `A`; since `A` is never assigned, a_0 is the value of an independent parameter passed to the procedure `P`. Thus a_0 happens to be both a dependent and independent parameter. The same can be said about b_0. The value of x_2 is the result of fetching from memory the contents of the variable `X` after the execution of the node "2".

BEAM normally does not evaluate dependent parameters from specific values of independent parameters. Instead the derivation of values goes in the opposite direction – given constraints on dependent parameters, are there values of independent parameters so that the constraints are satisfied? Section 4 discusses this issue, and can be considered a description of all that needs to be known about the data-flow portion of the graph, as well as the dependent and independent parameters.

Definition 1 *A **control graph** is a directed acyclic graph with the following properties:*

It has exactly one node without input edges, called ENTRY.
It has exactly one node without output edges, called EXIT.
Each edge has a predicate associated with it.
Each edge is labeled as admissible or inadmissible.
Some nodes are labeled as error, and if so, there is also a label indicating single or multi-statement criterion.

The predicate associated with an edge e, written **pred(e)**, expresses the condition of traversing the edge. For example, a statement of the form `if (A) ...` will give rise to two edges one for the then clause and the other for the else clause. They would be labeled with the predicates $a = 0$ and $a \neq 0$, respectively. (a is a dependent parameter, which is the result of fetching the contents of `A` just before the if statement.) In addition, edges where two paths meet are also labeled with their predicates. All other edges are shown without a label, which implies the always true predicate.

In the control graph of Figure 11 there is only one inadmissible edge, namely the one labeled x_2 `== NULL` (indicated by thick line); all other edges are admissible. There is only one error node, namely the one labeled ERROR (indicated by thick oval). Later we will consider both single- and multi-statement criteria applied to the error node.

Definition 2 *A **path from x to y**, where x and y can be nodes or edges, is a sequence of nodes and edges, such that*
x is the first element of the sequence,
y is the last element of the sequence,
every node in the sequence (other than x) is preceeded by one of its input edges,
every node in the sequence (other than y) is followed by one of its output edges,
every edge in the sequence (other than x) is preceeded by its source node,
every edge in the sequence (other than y) is followed by its sink node.

Definition 3 *If $x \neq y$ and there is a path from x to y then we say that x is **above** y and y is **below** x.*

The next two definitions relate to the possibility of a node y being an error node. We need to avoid reporting a path leading to y that would contain another error node.

Definition 4 *A path from x to y is **error-avoiding**, if it contains no error node, other than possibly y.*

Definition 5 *An edge d is a **dominator** of a node or edge y if d lies on every path from ENTRY to y.*
*An edge d is an **error-avoiding dominator** of y if d lies on every error-avoiding path from ENTRY to y.*

Dominators can be calculated in time linear in the number of edges [2].

4 Satisfiability

This section defines satisfiability of a set of predicates. Recall that predicates are expressed in terms of dependent parameters, which in turn are the results of some data operations on the independent parameters. This section does not describe algorithms for deciding satisfiability, only the interface through which the algorithms are accessed.

Definition 6 *A set of predicates is **satisfiable in a state-insensitive way** if there exist values for all the dependent parameters so that each predicate is true.*

Definition 7 *A set of predicates is* ***satisfiable in a state-sensitive way*** *if there exist values for all the independent parameters so that each predicate is true.*

State-sensitive satisfiability takes into account state changes (e.g., assignments), while state-insensitive satisfiability does not. For example, in Figure 1(a) an uninitialized value of `X` would be returned if the two predicates $\{a = 0, i \neq 0\}$ could be satisfied. Here a and i are two dependent parameters (the results of fetching the contents of variables `A` and `I`). The two predicates are satisfiable in a state-insensitive way, but not in a state-sensitive way.

Definition 8 *A path is* ***executable*** *if the set of predicates along the path is satisfiable in a state-sensitive way.*

In this paper we will assume that predicates have been assigned to edges so that the following property holds: Any executable path from ENTRY to any node can be extended to an executable path all the way to the EXIT node.

We will not discuss how to decide whether a set of predicates is satisfiable, but we will assume the existence of two kinds of **solvers** – a state-insensitive solver, denoted σ, and a state-sensitive solver, denoted Σ. A solver is a data structure, which at any time contains a set of predicates and has the following operations:

- A solver can be initialized to any set of predicates.
- A solver can be queried as to whether its set of predicates is satisfiable.
- Σ **implies** a predicate p if any parameter values satisfying the predicates of Σ also satisfy p.
- $\Sigma \cup p$ is a new solver obtained from Σ by adding the predicate p.
- $\Sigma_1 \cap \Sigma_2$ is a new solver containing those predicates implied by both Σ_1 and Σ_2.
- $\Sigma(p)$ is a "simplification" of the predicate p under the conditions of Σ. We will not describe how the simplification is done. We will use it only in comparison, such as $\Sigma_1(p) = \Sigma_2(p)$.

While the above description used the symbol Σ, the same operations also apply to a state-insensitive solver σ. The state-insensitive solver σ will be used to collect not just a set of predicates, but a sequence of edges carrying the predicates, which represents a partial path.

Example: In Figure 11 consider the state-sensitive solver initialized to the set of predicates a_0 == 0, b_0 == 0. Under these two conditions x_2 is `NULL`, thus the solver implies the predicate of the edge leading to the error node. In contrast, the state-insensitive solver initialized to the same predicates would not imply anything about x_2.

Example: In Figure 11 consider the state-sensitive solver Σ initialized to the single predicate b_0 == 0. Under this condition the contents of `X` after the second if-statement of Figure 6 is the same as it was after the first if-statement. Therefore $\Sigma(x_2)$ = x_1, where x_1 is the result of fetching the contents of `X` after node "1" in Figure 11.

For an edge e we will use the notation $\Sigma \cup e$ to mean $\Sigma \cup pred(e)$. Similarly we will use $\Sigma(e)$ and will talk about Σ implying an edge. For a set C of edges, $\Sigma_1(C) = \Sigma_2(C)$ means that $\Sigma_1(e) = \Sigma_2(e)$ for each $e \in C$.

In this paper we do not discuss our representation of the data operations because their handling is done by the solvers, whose implementation need not be described either. In addition, a state-sensitive solver, by its nature, understands any state changes and other operations on memory, which are represented by nodes in the control graph. Since the solvers completely take care of all memory accesses, the data flow algorithms below need not be concerned with them, and deal with the control flow conditions only.

5 Algorithms

In this section we assume a fixed graph (representing a procedure). First we present an algorithm for the multi-statement criterion and later for the single-statement criterion. The goal of both algorithms is to generate a path leading to an error, which is to be presented to the user. Both of these algorithms merely find partial paths that could lead to an error. Therefore, they are followed by a separate step (Section 5.3) that completes the partial path.

5.1 Multi-statement criterion

In this subsection we assume a fixed control node, called the **accused node**, which is marked as error and to which the multi-statement criterion is to be applied. Please recall that it is not sufficient to find an executable path leading to the accused node; that path must not contain any other error (Definition 9). In addition, it must be consistent with restrictions on any initial state of global memory, which are represented by admissible edges (Definition 10.)

Definition 9 *A path is **acceptable** if it is an executable path from ENTRY to EXIT, which does not contain any error nodes above the accused node.*

Please note that an acceptable path is not required to contain the accused node. For example, in Figure 11, any path from ENTRY to EXIT containing exactly one of the three nodes on the right is acceptable. But the path containing all three of those nodes is not acceptable, because it is not executable.

Definition 10 *A set of edges $e_1, \ldots, e_n$ is **evidence** against the accused node, provided*

1) Each edge e_i is admissible.

2) There exists an acceptable path containing all the edges $e_1, \ldots, e_n$ as well as the accused node.

3) Every acceptable path containing $e_1, \ldots, e_n$ also contains the accused node.

4) Each edge e_i is above the accused node.

For example, in Figure 11, the two edges labeled $a_0 == 0$ and $b_0 == 0$ constitute evidence against the error node.

Condition 4) is imposed because evidence below the accused node is difficult to convey to the user, for whom execution naturally terminates at the first occurrence of an error.

Definition 11 *An edge c is a **conspirator** of a node if it is an inadmissible error-avoiding dominator of the node.*

```
if (A)
  X = 0;

if (B)
  abort();
```

Figure 12: Value of A is irrelevant to the error.

For example, in Figure 11 the edge labeled x_2 == NULL is a conspirator of the error node. (It is the only conspirator.) Its predicate must be implied by some admissible edges before the error can be declared possible.

Notation: Let **C** be the set of all the conspirators of the accused node.

The algorithm of this section assigns a set of "bundles" to each node and edge above the accused node. A bundle is a pair (σ, Σ), where
σ is a state-insensitive solver, and
Σ is a state-sensitive solver.
Intuitively, σ will collect a sequence of edges defining a set of acceptable paths constrained to contain the accused node. Σ will contain the same set of edges, excluding any inadmissible edges. Since Σ has fewer predicates that σ, it will define a larger set of paths, some of which may avoid the accused node. If we can find a bundle where σ and Σ define the same set of paths, then the admissible edges of σ are good candidates for evidence against the accused node.

We always maintain any set of bundles in a normalized form:

Algorithm 1 *Given a set of bundles $\{(\sigma_1, \Sigma_1), ..., (\sigma_n, \Sigma_n)\}$* ***normalize*** *it by performing the following:*

- *Delete any (σ_i, Σ_i) if σ_i or Σ_i is unsatisfiable.*
- *Delete any (σ_i, Σ_i) if σ_i or Σ_i implies that a conspirator of the accused node is false.*
- *If $\Sigma_i(C) = \Sigma_j(C)$ then delete both bundles i and j and replace them with $(\sigma_i \cap \sigma_j, \Sigma_i \cap \Sigma_j)$.*

Algorithm 1 deletes any bundle that cannot possibly represent an acceptable path to the accused node. It also defines what it means for two bundles to have the same relationship towards the accused node. By using the condition $\Sigma_i(C) = \Sigma_j(C)$ we are making a judgment not to keep track of the bundles i and j separately.

For example, consider Figure 13, which is the control graph for the program segment in Figure 12. (In Figure 13 a_0 and b_0 represent the contents of the variables A and B.) During the execution of Algorithm 2 on Figure 13 we would consider two bundles $(\sigma_1, \Sigma_1), (\sigma_2, \Sigma_2)$, where
σ_1 contains $a_0 = 0, b_0 \neq 0$,
Σ_1 contains $a_0 = 0$,
σ_2 contains $a_0 \neq 0, b_0 \neq 0$,
Σ_2 contains $a_0 \neq 0$.

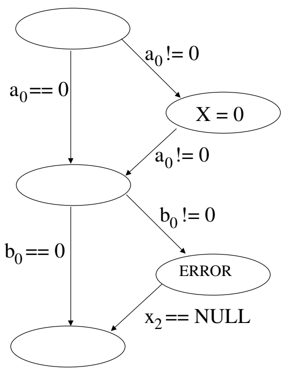

Figure 13: Graph of Figure 12

Algorithm 1 would merge these two bundles into (σ, Σ), where σ contains $b_0 \neq 0$, and Σ contains no predicate. Thus we have merged two bundles, each representing a single path into one bundle of two paths.

By merging two bundles we eliminate conditions that appear irrelevant to the error. This is a potential source of inaccuracy, but it is essential, because otherwise we would be enumerating all the paths.

Algorithm 2 *Given an accused node, this algorithm tries to find a bundle representing evidence against the accused node. The algorithm first computes a set of bundles for each node and edge by traversing the control graph from the accused node to the ENTRY node. Then it checks whether the ENTRY node has a bundle representing potential evidence against the accused node. If so, it uses Algorithm 4 to find in the bundle an executable path to be presented to the user.*

1) *Every edge e that is not above the accused node gets the empty set of bundles.*

2) *The accused node gets the result of normalizing the bundle* (σ, Σ)*, where*
σ is initialized to the error-avoiding dominators of the accused node, and
Σ is initialized to the admissible error-avoiding dominators of the accused node.

3) *Given a node with a set of bundles* $\{(\sigma_1, \Sigma_1), ..., (\sigma_n, \Sigma_n)\}$*, the set of bundles associated with an incoming edge e is calculated as follows.*

If an error node dominates e, then e gets the empty set of bundles.

Otherwise, if e is admissible then e gets the result of normalizing the set of bundles
$(\sigma_1 \cup e, \Sigma_1 \cup e), ..., (\sigma_n \cup e, \Sigma_n \cup e)$.

Otherwise (e is inadmissible) e gets the result of normalizing the set of bundles
$(\sigma_1 \cup e, \Sigma_1), ..., (\sigma_n \cup e, \Sigma_n)$.

4) *Given a node whose all outgoing edges have a set of bundles calculated, the node gets the result of normalizing the union of all the outgoing sets of bundles.*

5) *If one of the bundles (σ, Σ) calculated for the ENTRY node has the property that Σ implies all the conspirators of the accused node, then this algorithm succeeded. All the admissible edges of σ constitute potential evidence against the accused node; Algorithm 4 still needs to be called to see if an executable path can be generated.*

Please note that in step 5) there can be at most 1 bundle (σ, Σ), such that Σ implies all the conspirators. There reason is that if there were two, then Algorithm 1 would merge them into one.

Example: We will apply the algorithm to Figure 11. The accused node is the ERROR node. It has only two dominator edges: the outgoing edge of ENTRY, which is admissible, and the incoming edge of ERROR, which is inadmissible. Thus the error node gets the bundle (σ, Σ), where σ contains the predicate $x_2 ==$ `NULL` and Σ is empty.

The edge labeled $x_2 ==$ `NULL` and the node labeled "2" get the same bundle (σ, Σ).

The edge between node "1"and "2" gets the bundle $(\sigma \cup \{b_0 = 0\}, \Sigma \cup \{b_0 = 0\})$. Note that the state sensitive solver $\Sigma \cup \{b_0 = 0\}$ implies that $x_2 = x_1$, that is, under the condition that $b_0 = 0$ the contents of `X` after node "2" is the same as after node "1". This simplifies the predicate of the conspirator to $x_1 ==$ `NULL`.

For the edge between node "`X = &N`" and node "2" we calculate the bundle $(\sigma \cup \{b_0 \neq 0\}, \Sigma \cup \{b_0 \neq 0\})$. The state sensitive solver $\Sigma \cup \{b_0 \neq 0\}$ implies that $x_2 = \&N$, which makes the predicate $x_2 ==$ `NULL` of the conspirator false. Therefore Algorithm 1 will delete this bundle. As a consequence, the node "`X = &N`" as well as both of its edges get the empty set of bundles.

The node "1" gets all the bundles on both outgoing edges, which is the single bundle $(\sigma \cup \{b_0 = 0\}, \Sigma \cup \{b_0 = 0\})$.

The edge between node "`X = NULL`" and "1" gets the bundle $(\sigma \cup \{b_0 = 0, a_0 = 0\}, \Sigma \cup \{b_0 = 0, a_0 = 0\})$. Note that the state sensitive solver $\Sigma \cup \{b_0 = 0, a_0 = 0\}$ implies that x_2 = `NULL`. This simplifies the predicate of the conspirator to true.

For the edge between node "`X = &M`" and node "1" we calculate the bundle $(\sigma \cup \{b_0 = 0, a_0 \neq 0\}, \Sigma \cup \{b_0 = 0, a_0 \neq 0\})$. The state sensitive solver $\Sigma \cup \{b_0 = 0, a_0 \neq 0\}$ implies that $x_2 = \&M$, which makes the conspirator false. Therefore Algorithm 1 will delete this bundle. As a consequence, the node "`X = &M`" as well as both of its edges get the empty set of bundles.

The two remaining nodes, including ENTRY, get the single bundle $(\sigma \cup \{b_0 = 0, a_0 = 0\}, \Sigma \cup \{b_0 = 0, a_0 = 0\})$, which makes the conspirator true. Therefore all the admissible edges along the path, which correspond to $a_0 = b_0 = 0$, form potential evidence against the accused ERROR node.

Please note why it is important to perform backward analysis rather than forward analysis. If we had performed forward analysis, we would have encountered the condition $a_0 = 0$ first. But $a_0 = 0$ does not allow any simplification of the x_2, which would force us to propagate more bundles.

5.2 Single-statement criterion

The single-statement criterion adds an extra condition to Definition 10, namely that n = 1.

Algorithm 3 *The algorithm calculates a set of bundles for a given admissible edge e. It generates a bundle for every error node against whom the single edge e could serve as evidence. As in Algorithm 2 each resulting bundle needs to be passed to Algorithm 4 to find an executable path to be presented to the user.*

- *Let Σ be the state-sensitive solver consisting of all the error-avoiding dominators of e.*

- *For each error node x below e, that is labeled with the single-statement criterion, add to the result the bundle (σ, Σ), provided Σ implies all the error-avoiding dominators of x. The solver σ is set to contain all the dominators of e and x.*

Example: We will apply the algorithm to Figure 11. It has only one error node; any evidence against it would have to imply its dominator, namely x_2 == NULL. If we consider as potential evidence the edge "a_0 == 0", its solver Σ contains only the dominator "a_0 == 0", which does not imply x_2 == NULL. The same can be said about the edge "b_0 == 0" or any other of the edges. Therefore the empty set of bundles will be generated.

5.3 Establishing an executable path to error

This algorithm takes as input a bundle (σ, Σ) generated by Algorithm 2 or 3. The bundle represents a set of paths, namely those paths sharing all the edges in σ. This algorithms tries to find an executable path in that set. The solver σ can be thought of as a partial path and this algorithm then tries to complete the partial path into a full executable path.

For example, in case of Figure 13, σ would contain the predicate $b_0 \neq 0$, but no predicate constraining a_0. Algorithm 5.3 has to pick one of the two edges constraining a_0. Algorithms 2 and 3 are designed so that an arbitrary choice will usually result in an executable path.

Algorithm 4 *(We will describe the algorithm only informally.) It is a simple backtrack, at each stage adding to the original σ an input edge of a node already included in the path, until an executable path is constructed. The edge added to σ is also added to Σ in order to check consistency. If we need to chose an input edge of a particular node, but none would yield a satisfiable σ and Σ then we backtrack. In order to prevent exponential execution there is a constant, BacktrackLimit, that limits the number of backtracks. In the experiments of Section 6 the BacktrackLimit was 5.*

The initial solvers σ, Σ computed by Algorithms 2 and 3 are useful for two reasons. First they constrain the path to be generated, which makes the backtrack search practical. Secondly, the user will be given not only the path generated by this algorithm, but also the partial path σ, which represents those conditions that BEAM considers essential to the error. In contrast, the edges added by this algorithm are usually irrelevant to the error. (This algorithms still has to be used because Algorithms 2 and 3 do not guarantee that the partial path they construct can be extended to a full executable path.) Thus the information in σ is important for isolating the cause of the error.

5.4 Overall algorithm invocation

Algorithm 2 is applied potentially to a large number of accused nodes. That requires repeated forming of the initial Σ assigned to the accused nodes, which might take a large amount of the time. To reduce the time we note that one edge may dominate many error nodes; this edge will be added repeatedly to many state-sensitive solvers using the $\cup$ operation. The implementation of state-sensitive solvers is such that the $\cup$ operation is expensive, while all the other operations are not. This expense can be reduced by sharing it for many error nodes. This is done by the following algorithm that precomputes the state-sensitive solvers.

Algorithm 5 *For each node and edge x of the control graph calculate a state-sensitive solver Σ_x by propagating the solvers from ENTRY to EXIT.*

- *The ENTRY node gets the solver containing the empty set of predicate.*

- *Consider a node with a solver Σ and an outgoing edge e. If the node is marked as error then e gets an inconsistent solver. Otherwise, if e is admissible then e gets the solver $\Sigma \cup e$. Otherwise, (e is inadmissible) it gets the solver Σ.*

- *Consider a node with incoming edges $e_1, ..., e_n$ whose solvers are $\Sigma_1, ..., \Sigma_n$, respectively. The node gets the solver $\Sigma_1 \cap ... \cap \Sigma_n$.*

For each node and edge x, Algorithm 5 computes the solver Σ_x containing all the admissible error-avoiding dominators of x. Some nodes and edges get an unsatisfiable solver; those are nodes or edges representing dead code, or code that could be reached only after an error. The algorithm performs the expensive $\cup$ operation only once per edge.

All the algorithms of Section 5 are executed together as follows. Consider one step of Algorithm 5 just after Σ_x has been calculated for node or edge x. In case x is an error node labeled with the multi-statement criterion, then apply Algorithm 2; that algorithm can start with the just calculated Σ_x to be attached to the accused node x. In case x is an admissible edge then apply Algorithm 3; that algorithm can start with Σ_x after adding all the conspirators of x. (The conspirators need to be added because Σ_x contains only admissible edges.) As soon as Algorithm 2 or 3 generates a bundle representing potential evidence, it calls Algorithm 4 to generate an executable path to be reported to the user.

5.5 Worst case complexity

The most time consuming operation is the $\cup$ operation on a state-sensitive solver. Therefore we can measure worst case complexity by the number of times $\cup$ is performed. Let N be the number of all nodes and E be the number of edges.

Algorithm 5 performs the $\cup$ operation for each edge, that is, O(E) times.

Algorithm 2 does not need to perform a $\cup$ operation in step 2) because it uses the result of Algorithm 5. It performs the $\cup$ operation only in step 3), where it is performed once for each bundle. It is essential that the number of bundles be kept as small as possible; in fact, for practicality it needs to be kept constant independent of the graph size. We can assume that we impose some artificial upper bound on the number of bundles, to keep it constant, at the expense of failing to find some errors. In the implementation no limit was imposed because we never saw the number of bundles to exceed 4. Assuming a constant bound on the number of bundles, Algorithm 2 executes the $\cup$ operation $O(N \times E)$ times because it may be called on $O(N)$ error nodes, and for each it may execute step 3 $O(E)$ times.

Algorithm 3 needs to perform the $\cup$ operation for each edge e and for each conspirator of e. Since the number of conspirators is bounded by N, the worst case is again $O(N \times E)$.

If Algorithm 4 had no BacktrackLimit, then it might explore all paths, which would give it an exponential running time. However, a fixed BacktrackLimit limits the number of paths to the BacktrackLimit amount, which is then only a constant multiplier for worst case complexity. For one path, Algorithm 4 may execute the $\cup$ operation as many as N times. The algorithm itself may be called on the results of Algorithm 2 once for each error node, that is, N times. It may be called on the results of Algorithm 3, for each pair of an accused node and evidence against it, that is $O(N \times E)$ times. In total the worst case complexity of Algorithm 4 is $O(N^2 \times E)$.

The above worst case complexity estimation makes not only the assumptions that the $\cup$ operation dominates the running time, but also that it can be done in constant time independent of the size of the graph. The latter is not true in theory, however, in practice we do limit the state-sensitive solvers to a fixed amount of time, which gives them a constant upper bound. If a solver is not able to finish within the time limit then its conditions are considered inconsistent, which is conservative in the sense that it will avoid reporting bogus errors. (In the experiments in Section 6 the time limit was 1 second.) The consequences of that limit will be discussed in Section 5.6.

While the overall worst case complexity is $O(N^2 \times E)$, in practice its complexity is $O(N^2)$ for the following two reasons. First, Algorithm 4 has the $O(N^2 \times E)$ complexity because it could be invoked $O(N \times E)$ times on the results of Algorithm 3. In reality, it is very unusual for Algorithm 3 to find any potential evidence, therefore Algorithm 4 is normally run only for potential evidence found by Algorithm 2, which can happen $O(N)$ times. Secondly, the number of edges tends be linear in the number of nodes, thus any $O(N \times E)$ complexity can be replaced by $O(N^2)$.

We cannot expect better than $O(N^2)$ complexity because for each of N error nodes we may need to generate a path as long as N nodes. Conversely, we cannot afford anything worse than $O(N^2)$, and we had to impose some constant limits to achieve that.

5.6 Limitations

The algorithms of Section 5 are not exact, in that they may fail to find existing evidence against an error node, or conversely, they may report incorrect evidence. In this section we discuss under what conditions this may happen.

First, as mentioned at the end of Section 5.5, for efficiency reasons we cannot allow the state-sensitive solvers to do a perfect job of detecting inconsistency. When a state-sensitive solver runs out of time, it declares itself inconsistent; this may result in failure to detect errors, but will not result in reporting non-existent errors.

Another limitation of the solvers is that they do not have a complete understanding of arithmetic or other domains. The author has been giving the solvers only those axioms that were actually found lacking; too many axioms would slow them down. This process has converged for our application domain, for which it was found necessary to handle integer and pointer arithmetic with the operations $=, \neq, <, +, -$; the solvers have little information regarding multiplication and division, except where one of the operands is a constant. The solvers have practically no understanding of floating point arithmetic.

Secondly, Algorithm 4 may fail to find a path satisfying condition 2) of Definition 10 even if such a path exists. This has been observed to happen and the BacktrackLimit had to be adjusted so as to provide a good trade-off between the number of errors reported and running time. The goal is to maximize the number of errors detected for given amount of time.

Thirdly, the results of Algorithm 2 and Algorithm 3 are not guaranteed to satisfy condition 3) of Definition 10. As a result the user might be presented with an executable path leading to the error, although there is not enough evidence that the error will happen. This problem has never been observed, and it is difficult to construct a program where it could happen, but it is possible to construct an artificial control graph (Figure 14) to illustrate the possibility. In Figure 14, three edges are labeled with predicates p, q, r; these edges are inadmissible, while all other edges are admissible. The ERROR node has only one dominator, namely the edge of the ENTRY node; it has no conspirators. Algorithm 3 will identify the edge of the ENTRY node as evidence against the ERROR node, and Algorithm 4 will find an acceptable path leading to the ERROR node. But it is not true that every acceptable path will contain the ERROR node. This problem could not happen if all specifications of legal inputs were given in the form of asserts, say. Then the error-avoiding dominators would exactly characterize all the legal inputs.

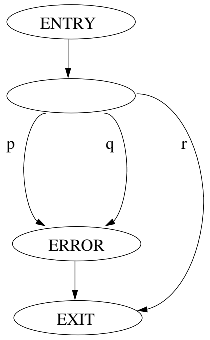

Figure 14: There is no evidence of error, but a path to error will be reported.

As described in Section 1, exact solutions could be guaranteed only if all procedure side-effects could be described exactly, if all procedure preconditions were given, and if consistency of a set of predicates could be decided.

6 Experimental results

This section reports on experience in using BEAM in our application area, which is design automation software. Table 1 lists four programs and for each it gives its size, time to process and number of errors found. All four programs have been through unit and system testing, and have been used for some time on experimental basis. The program labeled 42K does placement of circuits, the programs labeled 274K and 485K are two logic synthesis systems, and 46K is BEAM itself.

In Table 1, size is given in thousands of lines of code, which counts all the lines inside procedure bodies. This count includes comments and blank lines inside procedures, but does not include anything outside of procedure bodies.

We provide elapsed time rather than CPU time because it is more meaningful to the user; large amounts of time are spent in file IO. We give time for each of four phases of BEAM – "parsing" (using the parser of [16]), "graph" (building and simplification of the graphs), "inter-proc" (building of procedure summaries), and "DF analysis". The phase labeled "DF analysis" includes any inlining of procedure calls, simplification of the graphs, the algorithms discussed in this paper as well as other data flow algorithms not discussed.

All the errors in Table 1 are reported using the "medium policy" of Section 2.4, and are divided into three categories. Errors that do not require data flow analysis ("non DF") include the usual lint-like errors of unreachable statements, bad printf formats, etc. These errors are detected during the graph generation phase. Errors that do require data flow analysis are divided into those discussed in this paper ("described DF") and those detected by other data flow analyses ("other DF"); the later include uninitialized variables, memory leaks, etc.

In the program labeled 46K there were no errors found during this statistics collecting run, because BEAM is used on itself regularly. The programs with sizes 274K and 485K also had been analyzed by BEAM in the past; the errors reported this time reflect code changes, new feature of BEAM, and errors not corrected in the past. The "non DF" errors for 274K include some violations of project specific coding rules, not all of which would necessarily make the software fail.

SIZE	ELAPSED TIME IN MINUTES				NUMBER OF ERRORS FOUND		
	parsing	graph	inter-proc	DF analysis	non DF	described DF	other DF
42K	15	31	1	163	12	5	64
46K	9	60	2	397	0	0	0
274K	102	481	6	1707	1312	27	30
485K	469	765	12	3151	492	48	303

Table 1: Experimental results

7 Conclusions

We have presented algorithms for data flow analysis restricted to executable paths; these algorithms constitute a compromise between two extremes. One extreme is unrestricted data flow, which calculates one piece of information for ALL paths, without checking executability; this is too inaccurate. The other extreme enumerates all paths and checks executability separately; this is infeasible. Our approach enumerates bundles of paths, and keeps the number of bundles small (Algorithm 1). Although BEAM is required to generate a specific path leading to error (Algorithm 4), that algorithm needs to be called only when there is a good reason to believe that it will succeed and when candidate paths have been severely restricted by Algorithms 2 or 3.

We do not attempt to find all errors, but rather to maximize the number of errors found in a fixed amount of time. Section 6 reports on our experiences in this area. The time spent on data flow analysis of various kinds is several minutes per thousand lines of code, and the kind of analysis described in this paper detects errors at a rate of about one per ten thousand lines of code.

The success of this approach depends on, practical running times, reporting only what users consider real errors, and the handling of existing programs without requiring any specifications.

8 Acknowledgments

The author is grateful to R. Puri, L. Stok, L. Trevillyan, T. Kutzchebauch, J. Field, and G. Ramalingam for valuable discussions and reading of the manuscript. Special thanks go to Guido Volleberg who worked on inter-procedural analysis.

References

[1] L. D. Fosdick and L. J. Osterweil, "Data flow analysis in software reliability", *ACM Computing Surveys*, vol. 8, pp. 305–330, September 1976.

[2] A. Aho and J. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, MA, 1977.

[3] M. Wegman and F. K. Zadeck, "Constant propagation with conditional branches", *ACM Transactions on Programing Languages and Systems*, vol. 13, pp. 181–210, April 1991.

[4] J. Field, G. Ramalingam, and F. Tip, "Parametric program slicing", *in Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, pp. 379–392. ACM, 1995.

[5] M. Weiser, "Program slicing", *IEEE Transactions on Software Engineering*, vol. 10, pp. 352–357, July 1984.

[6] F. Tip, “A survey of program slicing techniques”, *Journal of Programming Languages*, vol. 3, 1995.

[7] G. Snelting, “Combining slicing and constraint solving for validation of measurement software”, *in Procedings of the Third International Static Analysis Symposium*, pp. 332–348, Aachen, Germany, September 1996. Lecture Notes in Computer Science.

[8] R. E. Strom and D. M. Yellin, “Extended typestate checking using conditional liveness analysis”, *IEEE Transactions on Software Engineering*, vol. 19, pp. 478–485, May 1993.

[9] J. Bradfield, *Verifying Temporal properties of systems*, Birkhauser, 1992.

[10] E. M. Clarke, O. Grumberg, and D. E. Long, “Verification tool for finite state concurrent systems”, in J. W. deBakker, W. P. deRoever, and G. Rozenberg, editors, *A decade of cuncurrency: Reflections and Perspectives, number 803 in Lecture Notes in Computer Science*, pp. 124–175. Springler Verlag, 1993.

[11] K. McMillen, *Symbolic Model checking*, Kluwer Academic Publishers, The Netherlands, 1993.

[12] B. Steffen, “Data flow analysis as model checking”, in Meyer A., editor, *Theoretical Aspects of Computer Software: TACS’91, volume 526 of Lecture Notes in Computer Science*. Springler Verlag, 1991.

[13] D. A. Schmidt, “Data flow analysis as model checking of abstract interpretations”, *in Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pp. 38–48. ACM, 1998.

[14] D. Evans, J. Guttag, J. Horning, and Y. M. Tan, “Lclint: A tool for using specifications to check code”, *in Proceedings of the SIGSOFT Symposium on the Foundations of Software Engineering*. ACM, December 1996.

[15] *PC-lint/FlexeLint 7.5*, Gimpel Software, 3207 Hogarth Lane, Collegeville, PA19426, USA, 1998.

[16] M. Karasick, “The architecture of Montana: An open and extensible programming environment with an incremental C++ compiler”, *in Proceedings of the 6th International Conference on the Foundations of Software Engineering*, pp. 131–142, November 1998.