RC24103 (W0610-052) October 11, 2006
Computer Science

IBM Research Report

Evidence-Based Analysis and Inferring Preconditions for Bug Detection

Daniel Brand
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Marcio Buss
Columbia University
New York, NY

Vugranam C. Sreedhar
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Evidence-Based Analysis and Inferring Preconditions for Bug Detection

Daniel Brand[1], Marcio Buss[2] and Vugranam C. Sreedhar[3]

[1] IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, USA,
danbrand@us.ibm.com
[2] Columbia University, NY, USA,
marcio@cs.columbia.edu
[3] IBM T.J. Watson Research Center, Hawthorne, NY, USA,
vugranam@us.ibm.com

Abstract. In order to gain user acceptance, a static analysis tool for detecting bugs has to minimize the incidence of false alarms. A common cause of false alarms is the uncertainty over which inputs into a program are considered legal. In this paper we introduce evidence-based analysis to address this problem. Evidence-based analysis allows one to infer legal preconditions over inputs, without having users to explicitly specify those preconditions. We have found that the approach drastically improves the usability of such static analysis tools. In this paper we report our experience with the analysis in an industrial deployment.

1 Introduction

There are several kinds of software tools for detecting program errors — from simple lint-like tools [1] to sophisticated software verification tools [2–8]. Each software tool has different goals and purposes, but from software developers perspective it is important to address two important problems for gaining wider acceptance of such tools:

Minimizing False Alarms: One thing that turns off software developers from using static analysis are false alarms. A false alarm, also called false positive, is an alarm (or a message) issued by a tool that user chooses not to translate into a code change. False positives occur when a tool has imprecise information about the given program and then chooses to issue a warning in spite of its uncertainty. False positives need to be traded-off against false negatives, which occur when the tool remains silent in the face of uncertainty. Therefore improving precision allows any tool to issue more true positives while reducing the false positive rate. There are several causes of imprecision that can lead to false positives, including lack of information about side-effects of library functions, lack of information about which inputs are considered legal, and imprecise analysis.

Minimizing Program Specification: The second thing that most developers dislike or forget to do is to write explicit specifications. Most sophisticated verification tools rely on such explicit specifications, which typically consist of preconditions, post-conditions, invariants, environments, etc. Absence of specifications results in either false positives or false negatives, or both. So it is important for tools to improve their accuracy in face of very little or no specifications.

In this paper we present *evidence-based analysis* for addressing the above two problems. We will show how to infer from a given source code which inputs are legal by focusing on what would be the programmer's intended preconditions for a procedure if they were written down explicitly. By inferring preconditions using various pieces of evidence in the program source code we will show how to reduce the number of false positives. The proposed approach as well as its implementation are language independent and for the rest of the paper we will use the C language for illustrating our examples.

To motivate our evidence-based analysis consider the example of Figure 1(a). Suppose we have no documentation for `foo()`, and know nothing about the potential callers of `foo()`. Should the programmer be warned of `foo()` failing if called with NULL argument? In other words, if `foo()` fails, is it because NULL is a legal input, and `x` should be checked for NULL before dereferencing? Or is the fault with the caller, because NULL is not legal input into `foo()`? There is no universally correct answer. It may differ from project to project depending on their reliability requirements and coding practices, or even the taste of individual programmer.

```
int foo(int *x)        int foo(int *x)
{                      {
  return *x;             if (x != NULL) bar();
}                        return *x;
                       }

(a)                    (b)
```

Fig. 1. Would you want a warning about possibly dereferencing NULL?

Now consider the example shown in Figure 1(b) and assume that a programmer is going through a manual code-review process with a code inspector.
Code inspector: "The procedure will fail if `x` is NULL."
Programmer: "The procedure is not to be called with `x` NULL."
Code inspector: "So why are you testing it for NULL?"
At this point the programmer will lose because most people agree that the code gives the impression that the procedure can handle NULL argument. We formalize this by assigning "admissible evidence" to the predicates on both the then- and else-branches of the if-statement.

To summarize, the main contribution of this paper is a framework for inferring legal program inputs from program text, and showing how to parametrize the process to match coding practices of individual projects. The rest of the paper is organized as follows. Section 3 defines *falsification conditions* and relates them to verification conditions. Section 4 explains how different predicates along a program path contribute different levels of evidence. Section 5 then explains how evidence based analysis impacts interprocedural analysis. Finally Section 6 shows benchmark results and discusses the method's use in an industrial deployment.

2 Related work

The problem of legal inputs was first tackled in the area of verification, also called "sound" analysis [2–8]. The goal in verification is the absence of false negatives, even to the point of providing a formal proof that certain properties are guaranteed to be satisfied. These approaches assume conservatively that (unless explicitly specified otherwise) all possible inputs can occur. An error is reported if all the conditions along a path to a potential error are satisfiable. For instance, LCLint[9] offers special notation to identify pointers that cannot be NULL; the default is the assumption that they can be NULL. Therefore without the extra annotation the programmer would receive a warning about Figure 1(a).

In contrast to verification, there are static analysis approaches that do try to infer preconditions so as to avoid false positives. FlexeLint[10] would flag the dereferencing of `x` in Figure 1(b), but it would not do so in Figure 1(a). The reason is that the if-statement suggests that the programmer intended to handle the possibility of `x` being NULL, but there is no such suggestion in Figure 1.

A more general approach, and the closest to ours, is [11], where this situation is referred to as contradiction in beliefs. While writing the if-statement the programmer believed that x could be NULL, but in the return-statement he believed it could not be NULL. In general, language constructs, e.g. if-statements or pointer dereferencing, generate beliefs, which are then propagated through the rest of the code. Any contradictions with other belief-generating constructs are flagged as potential errors.

Our approach has somewhat different goals. Reducing false negatives, which is the main motivation of [11], is only a side benefit for us. Our main goal is reducing false positives, and therefore our approach is execution-path oriented rather than based on the parse tree. Beliefs, or what we call levels of evidence, do not propagate syntactically. Instead they remain at their generating construct and we rely on a theorem prover to relate them along each path. This leads to a greater reduction in both false positives and negatives. In addition, the availability of a theorem prover [12] allows us to attach evidence to arbitrary predicates, not just finite-state properties as in [11].

Any method for inferring preconditions can also be used to infer post-conditions of unknown library functions. Additional pre- and post-conditions of library functions can be inferred from temporal behavior [11, 13, 14]. They observe patterns such as ordering of procedural calls (procedure `A` should be called before procedure `B`), or pairing of program statements (`lock` and `unlock` have to be paired). Such patterns are captured using finite-state machines and then used in typestate analysis [15–17, 8]. Our approach does necessitate modifications to typestate and other kinds of analysis, but those modifications are beyond the scope of this paper.

3 Falsification Condition

Software verification techniques are based on the following premise: given a specification of a program, a verification technique attempts to prove the *absence of errors* with respect to the specification. A failure is an indication of an error in the program or in the specification. In evidence-based analysis we have a different goal: prove the *presence of errors* without requiring specifications. For that we define falsification conditions and relate them to the verification conditions used in verification.

A program is a set of global variables and procedures defined in the usual way. Each procedure is represented by a control flow graph. Nodes in the control flow graph represent program statements, and edges represents flow of control from one node to another. Edges have labels representing conditions that need to be satisfied in order for the edge to be traversed. We insert special ERROR nodes in the graph where an error should be reported. For example, Figures 2(a) and (b) show the ERROR nodes representing NULL dereference for Figures 1(a) and (b). Interprocedural control flow graph also contains edges from procedure call to procedure definition, and edges from return statement back to the call statement. (We will discuss interprocedural analysis later in Section 5.) A *path* in a graph through a program is an alternating sequence of nodes and edges, and a path may traverse several procedures.

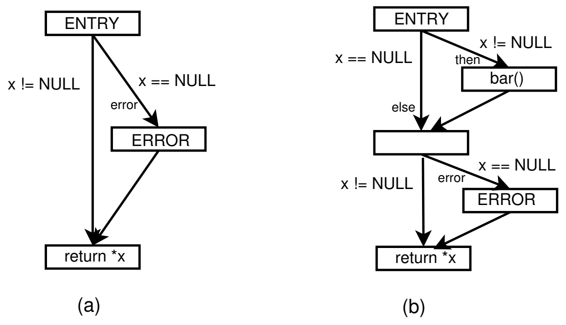

Fig. 2. Graphs for the examples of Figure 1

We now briefly explain verification conditions as used in verification to help the reader relate them to falsification conditions. In verification, only finite paths are considered; loops are either cut, or abstracted away. Each path P is assumed to have two predicates — $precondition(P)$ and $postcondition(P)$. The precondition is in terms of special symbols representing the initial values of variables; in contrast, the postcondition as well as all the expressions appearing along the path are in terms of program variables. The first step in generating a verification condition [18, 19] replaces values of program variables with their contents, which are symbolic expressions in terms of the initial values of variables. The easiest way of doing it is symbolic execution [20, 19] along the path. As a result, each edge e has attached a predicate $pred_e$, which is a boolean expression in terms of the initial values of variables, denoted $\mathbf{x}$. Likewise the $postcondition(P)$ is expressed in terms of $\mathbf{x}$.

Definition 1. *The* verification condition of a path P is valid *iff for all initial values* $\mathbf{x}$ *the post-condition must be true whenever the precondition together with all the predicates on the path are all true. That is,*

$$\forall \mathbf{x}\{(precondition(P) \wedge \bigwedge_{e \in P} pred_e) \Rightarrow postcondition(P)\}$$

A special case of a postcondition arises for a path P terminating in an ERROR node with incoming edge $error$. Then $postcondition(P)$ is the negation of $pred_{error}$.

And the path P has a valid verification condition if

$$\forall \mathbf{x}\{(precondition(P) \wedge \bigwedge_{e \in P'} pred_e) \Rightarrow \neg pred_{error}\}$$

where P' is the path P without the edge *error* and its ERROR node.

A valid verification condition implies that the ERROR node cannot be reached. By negation, the ERROR node is reachable if

$$\exists \mathbf{x}\{precondition(P) \wedge \bigwedge_{e \in P'} pred_e \wedge pred_{error}\} \tag{1}$$

Now we introduce *falsification condition*, which is also associated with a path. A valid falsification condition implies the reachability of an error when the path is exercised with *valid* inputs. The restriction to valid inputs has to be accomplished without knowing any precondition, which if present would define the valid inputs. We do it by extracting information from the source code; next section describes how to do it for various language constructs. The information is represented by associating "evidence", $evid_e$, with each edge e along the path. Each $evid_e$ is one of three values *admissible, inadmissible, asserted.*

The terminology comes from legal analogy. In an effort to convict the accused (*an error site*), prosecution asserts certain allegations (*reachable with legal values*), and then calls witnesses (*edges along particular path*). The statement made by each witness is deemed by the court admissible or inadmissible. In order to convict the accused the prosecution must collect enough admissible evidence to imply the asserted allegations (*condition (3)*). Inadmissible evidence plays no role at this stage; the prosecution cannot use it to its advantage, but does not need to prove it either. But even the inadmissible evidence cannot be completely ignored; if it contradicted other statements made by the prosecution side then the defense could take advantage of it. The accused can be convicted only if all the admissible and inadmissible evidence is consistent with the asserted accusations (*condition (2)*).

Definition 2. *The* falsification condition of a path P is satisfiable *iff all the predicates along the path are satisfiable. That is,*

$$\exists \mathbf{x}\{\bigwedge_{e \in P} pred_e\} \tag{2}$$

Please note that satisfiability for a falsification condition (2) is identical to (1) (in the absence of any precondition). In contrast to verification, however, satisfiability is not sufficient for reporting an error. The following additional condition is also required.

Definition 3. *The* falsification condition of a path P is valid *iff it is satisfiable and all the admissible predicates imply all the asserted predicates. That is,*

$$\forall \mathbf{x}\{(\bigwedge_{\substack{e \in P \\ evid_e = admissible}} pred_e) \Rightarrow (\bigwedge_{\substack{e \in P \\ evid_e = asserted}} pred_e)\} \tag{3}$$

Example: Consider Figure 2(b). Let $then, else, error$ be names of the three edges representing the then-branch of the if statement, the else-branch of the if statement, and the input into the ERROR node, respectively. Then these are the attached predicates and evidence.
$pred_{then}$ is `x != NULL`, $evid_{then} = admissible$
$pred_{else}$ is `x == NULL`, $evid_{else} = admissible$
$pred_{error}$ is `x == NULL`, $evid_{error} = asserted$
The path containing the edges *then* and *error* does not have a satisfiable falsification condition. But the path containing *else* and *error* has a valid falsification condition. Therefore that path would be reported to the user.

Consider Figure 2(a). There is only one path containing the edge *error*. That path has only the asserted predicate `x == NULL`. This makes the falsification condition satisfiable (2), but not valid (3). Therefore that path would not be reported to the user.

4 Assigning Evidence

In the previous section we explained the notion of a valid falsification condition, which depends on an assignment of evidence to the edges along a path. In this section we explain how to perform the assignment. It is based on a common observations that it is unusual for programmers to intentionally write useless code. From that we reason along the following lines. Given a predicate and given a program entry we ask the question whether modifying the predicate would change the behavior of the program when invoked from the given entry. The answer is an indication of how useful is the predicate, which gives us information about what inputs are expected at that program entry.

Consider an edge e with a predicate p, and we wish to assign evidence to the edge. Let S be the set of all the possible input values (legal or not) into the given program entry for which the edge e will be traversed with p being true. If we modify p into some p' the set S may change into S'. Assume that the modification has the property that S' is a strict subset of S. Then we ask the question "Did the behavior over the set of *legal inputs* change?" If the answer is "yes" then the set difference $S - S'$ must contain a legal input, and we assign admissible evidence to the edge e.

In the example of Figure 2(b) let's assign evidence to then-branch of the if-statement, which has the predicate `x != NULL`. The set S consists of all non-NULL addresses. if we modify `x != NULL` into `false` then S' will be empty. We would then expect the answer to the question "Did the behavior over the set of *legal inputs* change?" to be "yes". We would expect it on the assumption that the programmer did not intentionally write useless code. For that reason the then-branch of the if-statement is assigned admissible evidence. We can perform the same exercise for the else-branch, which would call for replacing the if-statement with `if(true)`, and likewise yield admissible evidence for the else branch.

Now suppose the answer to our question is "No, the behavior did not change over the set of legal inputs". And suppose that the answer continues to be negative even if the predicate p is replaced with the always false predicate. That means the predicate p does not provide any information about legal inputs, and the edge is useless. Then we ask the next question: "Is that useless edge intentional". If the answer is "yes" then the programmer believes that the edge cannot be traversed for any legal inputs. Telling him otherwise would not be productive unless we have evidence to support that claim;

therefore the edge e gets asserted evidence. If the edge is useless unintentionally then we are free to traverse it on a way to an error without having to prove that it is possible – the edge e gets inadmissible evidence. Later in the section we will show examples of language constructs generating inadmissible and asserted evidence, respectively.

The answers to our two questions depend on reliability requirements, coding standards of an individual project and human psychology. We want to emphasize that these considerations are part of daily software development process and are not an artifact of an automated tool. Therefore consider the common situation of a programmer having a dialog with a code inspector who is criticizing the programmer's code. (You have encountered them already in the introduction.) The inspector found a satisfiable path to an ERROR node and wants to argue that the path is exercisable by legal inputs. The programmer argues back that those inputs should not be considered legal. The dispute is decided by the programmer's colleagues, who may have to modify his code in the future. Therefore they want to make sure the code works for all inputs that appear legal to most people examining the code. They side with the inspector by making predicates admissible, and they side with the programmer by making them inadmissible or even asserted.

4.1 ERROR node

ERROR nodes are generated in the process of translating given source code into the control flow graph. For example, `if(x == NULL) abort()` is represented by an ERROR node whose incoming edge has the predicate `x == NULL`. Many languages have an `assert(c)` construct, which may be a macro expanding into `c ? 0 : abort()`; this is represented by the same ERROR node structure. Anybody writing such a constructs expects the code to be unnecessary, and he does it intentionally. He would admit that his code is wrong only if there is other information implying that the `abort()` call is reachable with legal inputs. That is expressed by assigning asserted evidence to the incoming edge.

As another example consider `return *x`. That could be rewritten as `if(x == NULL) abort(); return *x` to express the possibility of error. Although such a transformation would be performed by an automatic translation form original source code, we can still ask the question of whether it is expected to be dead code. And if the answer is "yes" then the evidence associated with that generated then-branch is asserted.

4.2 If and Switch Statements

In the introduction we explained why the then- and else-branches of an if-statement should be assigned admissible evidence. Now consider the example of cascaded if-statement in Figure 3(a).
Code inspector: "The index i will exceed the range of the array a, if `i > 2`."
Programmer: "The procedure is not to be called with `i > 2`."
Code inspector: "So why do you have the last if-statement?"
Programmer: "For clarity."
Code inspector: "Clarity would be better served by an assert."

It is difficult even for people to decide what the programmer meant. Incidentally, an identical situation occurs for switch statements with omitted default clause. People

```
int a[3];
int is_i_out_of_range(unsigned i)
{
  int r;

  if (i == 0) r = 0; else
  if (i == 1) r = 1; else
  if (i == 2) r = 2;

  return a[i];
}
```

(a)

```
int a[3];
int is_r_uninitialized(unsigned i)
{
  int r;

  if (i == 0) r = 0; else
  if (i == 1) r = 1; else
  if (i == 2) r = 2;

  return r;
}
```

(b)

Fig. 3. Should an error be reported in case i is none of 0, 1, 2?

who side with the code inspector would expect the program's behavior to change by replacing `if (i == 2)` with `if (true)`. Therefore they would assign admissible evidence to the unexpressed else branch of the last if-statement (or the missing default branch in a switch statement).

Those who side with the programmer think it possible that the behavior would not change by the above modification. The project may even have a coding guideline according to which such a cascaded if-statements implies an assertion that all cases are considered. If so, the programmer wrote the unnecessary code intentionally to express the assertion; therefore the final else branch would receive asserted evidence. On the other hand, if the programmer wrote the unnecessary code without intending such an assertion, the final else branch would receive inadmissible evidence.

People who would assign the inadmissible evidence are in a seemingly contradictory position where Figure 3(b) contains an error, while Figure 3(a) does not. In Figure 3(b) the uninitialized use of `r` would be flagged because there is no asserted evidence along the path containing the unexpressed else-branch. It seems contradictory because both functions would fail under identical assumptions about the parameter `i`. To understand the reasoning, imagine that the cascaded-if statement is removed from both procedures. Then Figure 3(b) has an obvious error, which reintroduction of the cascaded-if statement does not eliminated completely. In contrast, Figure 3(a) would have no error worth reporting and reintroduction of the cascaded-if statement does not change anything.

As you can see, cascaded if-statements can be controversial with all three level of evidence being candidates for representing a coding guideline. In our implementation there is a parameter to express the decision. The tool is shipped with the admissible setting for the last unexpressed else-branch of a cascaded if. Independently the user can turn on a warning for switch statements without a default clause which do not exhaust all cases.

4.3 Loops

Consider the example of Figure 4(a).
Code inspector: "The procedure will divide by 0 if `n == 0`."

```
int loop1(unsigned n)
{

  int i;
  for (i=0; i<n; i++)
  {

   /* anything */
  }
  return 42/n;
}
```

(a)

```
int loop2(unsigned n)
{
  int x = 0;
  int i;
  for (i=0; i<n; i++)
  {

   x++;
  }
  return 42/x;
}
```

(b)

```
int loop1(unsigned n)
{

  int i;
  for (i=0; ; i++)
  {
   if (i >= n) break;
   /* anything */
  }
  return 42/n;
}
```

(c)

Fig. 4. Is division by 0 possible?

Programmer: “It is not to be called with `n == 0`.”
Code inspector: “So why are you comparing it against 0 as the loop is entered?”
Programmer: “Because the syntax of the for-statement compels me to do so.”

The identical conversation, but with different outcome might occur for the example of Figure 4(b). In the eyes of his colleagues the programmer is likely to win the first debate, but he is likely to lose the second. The reasons are similar to the above example of cascaded if-statement. Imagine both examples with the loops removed. In Figure 4(a) there is no evidence for the condition of error, while Figure 4(b) would be guaranteed to fail. Simple reinsertion of the for-loop into Figure 4(a) does not give us any information about `n`, and therefore the presence of the loop should have no effect. Reinsertion of the loop into Figure 4(b) does not completely prevent the failure, which makes a warning acceptable.

It is important that our formalization allows the above sentiment about loops. The key is the level of evidence assigned to the then- and else-branches of the loop condition `i < n` on different iterations. It can be accomplished by assigning inadmissible evidence to the false-branch of the loop condition the very first time it is evaluated, while assigning admissible evidence thereafter. The programmer cannot argue that we have no evidence that the loop condition can ever be true or ever false. If he was sure that it can never be true then he would not write the loop, and if he was sure that it can never be false then he wrote an infinite loop.

However, the programmer could object to the following situation. Suppose the program fails only if the loop is executed exactly five times. We have no more evidence that it can be executed exactly five time than we have that it can be executed exactly zero times. Users who feel that way can assign inadmissible evidence to all the possible outcomes of the loop condition.

In our implementation, by default, inadmissible evidence is assigned only to the possibility of zero iterations, and that does not cause any problems with our users. However a more complicated situation exists with if-statements inside a loop. For example, Figure 4(a) could be rewritten as in Figure 4(c). As above, that should not offer any evidence for `n == 0`. This is in contrast to an if-statement `if (0 >= n) ...` which does provide such evidence.

In our implementation an if-condition inside a loop provides admissible evidence only if it does not involve any variables modified by the loop. This is conservative and is a source of false negatives, but it is necessary to avoid false positives.

4.4 Relational Operators

Consider the following example.

```
int a[10];
int foo(unsigned int i)
{
  if (i <= 10) return a[i];
  else         return 0;
}
```

Consider replacing `if (i <= 10)` with `if (i < 10)`. In the minds of many people that would make a difference, which means that the programmer expected `10` to be a possible value for `i`. That would make `i` exceed the range of the array `a`. Therefore admissible evidence can be assigned not only to `i <= 10`, but also to `i == 10`; that is important because the latter does imply that `i` will be out of range, but the former does not.

In our default setting, the if-statement is interpreted to provide three branches (instead of just two) – `i < 10`, `i == 10`, `i > 10` – all three with admissible evidence.

4.5 Correlated Input Variables

Consider the following example.

```
/* This procedure can handle n1 == 0 as well as n2 == 0 */
int average(int sum, unsigned int n1, unsigned int n2)
{
  return sum/(n1 + n2);
}
```

Code inspector: "The procedure will fail if `n1 == 0` and `n2 == 0`."
Programmer: "I promised to make it work if either is 0, but not if both are 0."
Both points of view have their supporters, but there are enough people who feel that even they would be misled the same way the inspector was. Therefore we have not implemented the programmer's point of view. That point of view cannot be formalized by assignment of evidence; it would require redefining the validity of a falsification condition by requiring that just a single admissible evidence is sufficient to imply all the asserted predicates along the path.

4.6 Procedure Calls

Consider the following example.

```
/* This procedure can handle n1 == 0 */
int average1(int sum, unsigned int n1, unsigned int n2)
{
  return average2(sum, n1, n2);
}

/* This procedure can handle n2 == 0 */
int average2(int sum, unsigned int n1, unsigned int n2)
{
  return sum/(n1 + n2);
}
```

Code inspector: "Both procedures will fail if `n1 == 0` and `n2 == 0`."
Programmer: "I did not promise for either one to work if both parameters are 0."

This time the programmer will get very wide support from his colleagues. The difference from Section 4.5 is that the two pieces of evidence are in different functions. There may be many contexts where it is OK to call `average2()` with `n2 == 0`, but that does not imply anything about `average1()`.

When a programmer writes a comment like the one for `average2()`, he is not expressing a precondition, in the sense of limiting possible inputs. Instead he is telling us that at the entry to `average2()` we can create an edge with `n2 == 0` having admissible evidence, which can be used to catch errors like dividing by `n2` itself. However, the evidence is admissible only for paths starting with `average2()`. The comment does not imply anything about legal inputs of `average1()` or any other procedure calling `average2()`. Therefore we cannot use `n2 == 0` as admissible evidence for paths starting at `average1()`. This phenomenon happens in situations other than procedure calls. Macros, like procedures, may be prepared to handle certain unusual parameter values, but that does not imply that every procedure invoking the macro is obligated to handle the same unusual values. It also happens in languages, like PL/I, allowing multiple procedure entries. One entry may declare it possible that `n1 == 0` and another that `n2 == 0`. That does not imply that it is wrong to divide by `(n1 + n2)` in their common code.

Let's assign evidence to an edge inside `average2()` labeled with `n2 == 0`. As above, we ask the question whether the behavior will change if we change the predicate to always false. The answer is clearly "yes" for any path starting at the entry to `average2()`, due to the comment. Therefore along such a path the edge would get admissible evidence. But for a path starting at the entry to `average1()` there is nothing to suggest that it could be invoked with `n2 == 0`. As far as `average1()` is concerned any testing of `n2 == 0` is useless. But it is not intended to be useless, therefore the evidence is inadmissible.

This example shows that evidence level cannot be fixed with each edge syntactically, but depends on a path. This is the general rule: No edge along a path can have admissible evidence unless the very first node on the path dominates the edge. Therefore any path starting at the entry of `average2()` can have admissible evidence on the edge representing `n2 == 0`. But paths starting at `average1()` cannot do so because the entry of `average2()` is reachable from elsewhere.

5 Interprocedural Evidence-Based Analysis

The purpose of inter-procedural analysis is to propagate information calculated in the body of one procedure to those calling it, as well as to those being called by it. As we will explain, evidence-based analysis diminishes the usefulness of information passed from callers to callees and increase the importance of information passed in the opposite direction. There are several reasons for this asymmetry.

Please recall Section 4.6, where we explained that admissible evidence in a caller can be used inside a callee, but not vice versa. In [11] this asymmetry is expressed by propagating beliefs (aka, admissible evidence) from callers to callees but not in the opposite direction. A contradiction in beliefs causes an error to be reported inside the callee. The method in [11] can do the propagation from caller to callee because they deal with just one piece of admissible evidence. This is not possible in the more general situation, where asserted evidence is implied by more than one piece of admissible evidence. During interprocedural analysis, before a specific path is fixed, we do not know which evidence can remain admissible. On the other hand, the situation is clear when propagating information from callee to caller – all admissible evidence needs to be down-graded to inadmissible or asserted. Error is then reported in the topmost caller appearing in the path; he should not have called the lower level procedure with values that will make it fail.

The main reason for the asymmetry is the fundamental philosophy of evidence-based analysis. The correctness of a software layer does depend on its specific lower level libraries, but it does not depend on any specific higher software layers. If higher or lower layers are available, information propagated from them can be used for two purposes – to determine that an error is possible, and to suppress invalid error complaints. To report an error only MUST information can be used, and that is rarely derived by interprocedural analysis. The more usual MAY information can be used for suppressing potential false positives. And here is a difference between lower and upper layers. Information derived from lower layers can override any evidence, because the lower layer is fixed. But information derived from upper layers cannot override any evidence, because the evidence can be used to imply an error that would happen with a different upper layer.

For example, consider the impact on pointer analysis. It is possible to report an error in a procedure while assuming that none of its parameters alias. The reason is that the very declaration of two separate parameters is admissible evidence that sometimes they do not alias. And that cannot be overridden by any pointer information derived from specific callers. For more complicated pointer expressions other forms of evidence may exist, and it continues to be true that information from specific callers cannot override it. The situation is different with information derived from lower layers. For example, the C-library function `strcpy` is such that after `x = strcpy(y,z);` the variables `x` and `y` alias. And the fact that the programmer declared two separate variables cannot override that. Therefore for evidence-based analysis, pointer information has to be propagated from callees to callers, but propagation from callers to callees is not essential. In order that such a pointer analysis be sound, information propagated from a callee to a caller needs to be adjusted for any aliasing present in the caller, but does not need to be adjusted for aliasing in a grand-caller.

For these reasons, our overall approach to inter-procedural analysis is to propagate information from callees to callers only, in the form of procedure summaries. There are several kinds of procedure summaries that we compute, including information for error reporting, MOD information, and other side-effects. These procedures summaries can be view as generated preconditions. If there is not enough evidence inside a procedure to report a potential problem, the summary describes assumed legal inputs. Callers are then checked to see if they pass illegal inputs. And if there is not enough admissible evidence inside the caller to prove that the inputs are illegal, the caller gets another summary expressing its legal inputs.

6 Experience

The method has been implemented in an IBM internal tool called BEAM [21], which currently handles C, C++ and Java. It went into production use in 2001 with usage steadily increasing since then. User acceptance is an indication that the tool reports useful errors with few false positives. One of the main reasons for that is the evidence-based analysis, which has evolved in response to user feedback. The tool is shipped with the default evidence settings described in Section 4. Users are able to change them, but we have not seen them do that.

The default settings are not perfect. We have seen false positives due to admissible evidence for cascaded-if and missing default as described in Section 4.2. Also we have seen false positives due correlated input variables (Section 4.5). However, users tend to tolerate those, because they feel that another person might misunderstand the code the same way as the tool. On the other hand, users do not tolerate false positives due to overly aggressive admissible evidence in a loop. They consider them bugs in the tool, and therefore we must be particular conservative in assigning evidence to predicates in a loop.

Table 1 indicates the impact of evidence level on the number of errors reported. Each of the benchmarks listed was run with various rules for assigning evidence, and for each the table shows the number of errors reported. (We count only those kinds of errors where evidence is used to decide whether to report them.) The meaning of the columns is the following.

ev-0: edges leading to ERROR nodes and edges inside callees have asserted evidence; all other edges have inadmissible evidence
ev-1: as ev-0, except then-branches of if-statements have admissible evidence
ev-2: as ev-1, except else-branches of simple if-statements have admissible evidence
ev-3: as ev-2, except else-branches of cascaded if-statements have admissible evidence
ev-4: as ev-3, except relational operators yield admissible evidence
ev-5: as ev-4, except eventual loop termination has admissible evidence
ev-6: as ev-5, except immediate loop termination has admissible evidence
ev-7: as ev-6, except pointer aliasing has admissible evidence
ev-8: as ev-7, except edges inside callees have inadmissible evidence
ev-9: as ev-8, except edges inside callees have admissible evidence
ev-all: all edges have admissible evidence; none has asserted evidence

From Table 1 we can see that the number of false positives would be intolerable (last column) if Definition 3 required paths to be only satisfiable (2) without condition (3). But the number of errors reported becomes tolerable even with the highest evidence setting, ev-9.

As an illustration of how our method works, consider a problem reported in firestorm-0.5.4 [4] in column ev-4 of Table 1. There is a function `mesg_initlog(tv, code, buf)` containing the expression `mesg_str[code]`. The subscript `code` is a parameter, and the array `mesg_str` is defined by to be of size 6 elements. The function `mesg_initlog` does not contain enough evidence to report the possibility of the subscript `code` being out of range. Therefore, instead, a precondition is generated

[4] A network intrusion detection system available from Free Software Foundation

Benchmark	ev-0	ev-1	ev-2	ev-3	ev-4	ev-5	ev-6	ev-7	ev-8	ev-9	ev-all
paraffins	0	0	0	0	0	0	0	0	0	0	8
compress	0	0	0	0	0	0	0	0	0	0	19
gzip	0	0	0	0	0	0	0	0	0	1	54
finger	0	4	6	6	6	6	6	6	6	7	175
perl	0	1	2	2	2	2	6	6	8	15	449
tar	0	9	15	15	15	14	16	16	17	17	520
bison	0	22	25	25	25	25	25	25	27	27	722
bash	0	7	11	11	13	16	15	16	17	24	814
firestorm	0	0	1	1	3	6	7	7	7	7	893

Table 1. Impact of evidence on number of errors reported

for the function `mesg_initlog`; the precondition describes legal values of the parameter `code`, namely `0 <= code < 6`. Then inside another function `mesg(code, fmt, ...)` there is a call `mesg_initlog(&tv, code, buf);`
In addition, there is a statement
`if ( code > M_MAX ) code=0;` which was probably meant to be
`if ( code >= M_MAX ) code=0;`
That caused our tool to report an index-out-of-range error, because the test `code > M_MAX` provides admissible evidence for `code == M_MAX`, see Section 4.4. Since `M_MAX` is defined to be 6, we have enough admissible evidence to falsify the generated precondition of `mesg_initlog(tv, code, buf)`.

7 Conclusions

We have presented an approach to determining that a program will fail in an intended invocation environment, without requiring any preconditions defining the intended environments. We have done that by introducing the notion of a falsification condition, whose validity implies the presence of error. This is a formalization of a common informal process by which programmers assign responsibility for checking unusual inputs between higher and lower layers of software. A falsification condition is parameterized by assignment of evidence levels to control flow edges, and we have described how to assign the evidence depending on a project's coding practices.

The approach reduces the incidence of false positives by reporting only those errors that happen for legal inputs. At the same time it decreases false negatives by reporting error that can happen in any intended environment, which may not even exist today. General user acceptance of our falsification-based tool is an indication that it leads to improved software quality.

Acknowledgments

The implementation of our method would not be possible without Florian Krohm, Francis Wallingford and Goh Kondoh, who also provided valuable suggestions for this paper.

References

1. Darwin, I.F.: Checking C programs with lint. O'Reilly (1988)
2. Hantler, S.I., King, J.C.: Introduction to proving the correctness of programs. ACM Computing Surveys **8**(3) (1976) 331–353
3. German, S.M.: Automating proofs of the absence of common runtime errors. In: Proceedings of the 5th ACM Symposium on Principles of Programming Languages, ACM (1978) 105–118
4. Jones, R.B., Dill, D.L., Burch, J.R.: Efficient validity checking for processor verification. In: Proceedings of the IEEE International Conference on Computer-Aided Design, Santa Clara, CA, IEEE (1995) 2–6
5. Windley, P.J.: Formal modeling and verification of microprocessors. IEEE Transactions on Computers **44**(1) (1995)
6. Crow, J., Owre, S., Rushby, J., Shankar, N., Srivas, M.: A tutorial introduction to PVS. In: Workshop on Industrial Strength Formal Specification Techniques. (1995)
7. Jacobs, B., van den Berg, J., Huisman, M., van Berkum, M.: Reasoning about java classes. SIGPLAN Notices **33**(10) (1998) 329–340
8. Fink, S., Yahav, E., Dor, N., Ramalingam, G., Geay, E.: Effective typestate verification in the presence of aliasing. In: Proceedings of International Symposium on Software Testing and Analysis. (2006) 133–144
9. Evans, D., Guttag, J., Horning, J., Tan, Y.M.: Lclint: A tool for using specifications to check code. In: Proceedings of the SIGSOFT Symposium on the Foundations of Software Engineering, ACM (1996)
10. : PC-lint/FlexeLint 7.5. Gimpel Software, 3207 Hogarth Lane, Collegeville, PA19426, USA (1998)
11. Engler, D., Chen, D., Chou, A.: Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In: Proceedings of Symposium on Operating Systems Principles. (2001) 57–72
12. Brand, D., Krohm, F.: Arithmetic reasoning for static analysis of software. Technical Report RC-22905, IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY 10598 (2003)
13. Ammons, G., Bodik, R., Larus, J.: Mining specifications. In: Symposium on Principles of Programming Languages, Portland, OR, ACM (2002) 4–16
14. Kremenek, T., Twohey, P., Back, G., Ng, A., Engler, D.: From uncertainty to belief: Inferring the specification within. In: Proceedings of the 7th Symposium on Operating System Design and Implementation. (2006)
15. Strom, R.E., Yellin, D.M.: Extended typestate checking using conditional liveness analysis. IEEE Transactions on Software Engineering **19**(5) (1993) 478–485
16. Strom, R.E., Yemini, S.: Typestate: A programming language concept for enhancing software reliability. IEEE Transactions on Software Engineering **12**(1) (1986) 157–171
17. Das, M., Lerner, S., Seigle, M.: Esp: Path-sensitive program verification in polynomial time. In: Proceedings of ACM Conference on Programming Language Design and Implementation, ACM (2002) 57–68
18. King, J.C.: A Program Verifier. PhD thesis, Carnegie-Mellon University, Pittsburg, Pennsylvania (1969)
19. K. Rustan, M. Leino, T.M., Saxe, J.B.: Generating error traces from verification-condition counterexamples. Science of Computer Programming **55** (2005) 209–226

20. King, J.C.: Symbolic execution and program testing. Communications of the ACM **19**(7) (1976) 385–394
21. Brand, D.: A software falsifier. In: International Symposium on Software Reliability Engineering. (2000) 174–185

8 Appendix

We show the relevant parts of the code from firestorm-0.5.4 discussed in Section 6.

```
#define M_MAX 6

char *mesg_str[]={"undefined","critical","error","warning","info","debug"};


void mesg_initlog(struct timeval *tv, int code, char *buf)
{
int ret;

if ( !init_buflen ) {
printf("warning: internal message buffer overflow\n");
return;
}

ret=snprintf(init_buf_p, init_buflen,
"%.8lu.%.6lu %s: %s\n",
tv->tv_sec, tv->tv_usec, mesg_str[code], buf);

init_buf_p+=ret;

if ( ret > init_buflen ) {
init_buflen=0;
}else{
init_buflen-=ret;
}
}



void mesg(unsigned char code, char *fmt, ...)
{
static char buf[1024];
struct timeval tv;
va_list va;

if ( code > M_MAX )
code=0;

gettimeofday(&tv, NULL);
va_start(va, fmt);
vsnprintf(buf, sizeof(buf), fmt, va);
va_end(va);

if ( !init_buflen ) {
/* No point checking for errors, what are we gonna do
 * print them to screen? ;) */
printf("%.8lu.%.6lu %s: %s\n", tv.tv_sec,
tv.tv_usec, mesg_str[code], buf);
fflush(stdout);
return;
}else{
mesg_initlog(&tv, code, buf);
}
}
```