RC 21788 (Log 98089) 30 June 2000 Computer Science

Research Report

A Software Falsifier

D. Brand
IBM Research Division
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

A Software Falsifier

Daniel Brand
IBM T. J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598, USA
brand@watson.ibm.com

July 5, 2000

Abstract

A falsifier is a tool for discovering errors by static source-code analysis. Its goal is to discover them while requiring minimal programmer's effort. In contrast to lint-like tools or verifiers, which try to maximize the number of errors reported at the expense of allowing "false errors", falsifier's goal is to guarantee no false errors. To further minimize programmer's effort, no specification or extra information about his program is required. That, however, does not preclude project specific information from being built in. The class of errors detectable without any specification is important not only because of the low cost of detection, but also because it includes errors of portability, irreproducible behavior, etc, which are very expensive to detect by testing. This paper describes the design and implementation of such a falsifier, and reports on the experience with its use for design automation software. The main contribution of this work lies in combining data-flow analysis with symbolic execution to take advantage of their relative advantages.

Keywords: static analysis, data-flow analysis, symbolic execution, theorem proving

1 Introduction

It is a common practice to complement software testing with source-code analysis. That analysis can be manual in the form of inspections and walk-throughs [1, 2], or automatic [3, 4, 5, 6] or it can be a combination of manual and automatic [7].

Source-code analysis offers several benefits. First, it can be performed before an executable version of the software exists. Secondly, some faults are more easily detected by source-code analysis than by testing, for example, memory leaks. Thirdly, it allows faster localization of the faults, because their symptoms tend to be closer to their cause.

For these reasons there are many tools automating source-code analysis. We will consider four types – compilers, lint tools, verifiers, and falsifiers; and we will consider only their fault detection capabilities, although some may have other purposes as well.

All such tools have the same ultimate goal of finding all faults. That goal is unattainable in general and therefore all approaches must compromise. Some tools compromise by not reporting all faults, others compromise by issuing "false errors", which do not correspond to any actual fault. A second important trade-off is between time and accuracy. If a tool spends more time it can do a more accurate determination of which potential faults could actually cause a failure during execution; this then allows it to detect more real faults and report fewer false errors.

The main use of a compiler is to produce an executable object code, but in this paper we consider only its ability to detect programming errors; for example, some compilers will flag uninitialized variables. Compilers tend to be the fastest among source-code analysis tools. The requirement of speed results in a great degree of uncertainty as to which potential faults are real. Compilers tend to resolve this uncertainty by reporting only what is certainly an error. Some compilers report only syntax errors, others look also for execution errors, which sometimes leads to false errors, but very few in comparison with lint.

Lint tools [4, 5, 6] perform the same type of analysis as compilers with about the same speed and accuracy. However, they do not try to avoid "false errors", which tends to be their major criticism. Among lint tools we include tools like [3], that may not call themselves "lint", but share the lint's two goals – speed, and maximizing the number of reported errors.

Verifiers (sometimes called formal verifiers) [8, 9, 10, 11, 12, 13] check a given implementation against a user supplied specification. They spend more time than the other source-code analysis tools, achieving the highest degree of accuracy. But there is still uncertainty for two reasons. First, the tool may not know which input combinations are considered legal, and secondly the problem may be too large for the verifier to handle. Both of these kinds of uncertainties are resolved by placing the burden of proof on the user; an error is reported if the user provided information does not allow the verifier to prove the absence of error.

In contrast to a verifier or lint, falsifier's goal is to guarantee than any reported error can actually cause a failure during execution. In this sense a falsifier is related to a compiler with the exception that falsifier is allowed more time so as to detect more difficult errors. (Our tool is meant to run overnight checking code written during the day.) A falsifier uses some of the same methods as a verifier, facing the same uncertainties, but resolving them by placing the burden of proof on itself. The price falsifiers pay for avoiding false errors is in missing some real faults.

There are three reasons why we emphasize avoiding false errors.
First, when faced with many false errors programmers simply stop using the tool.
Secondly, if they do not stop using the tool they may ignore some of its legitimate complaints. People have a tendency when looking at their own code to see what they meant to write, rather than what they actually wrote. It happens often that a programmer assumes the tool is mistaken when his errors are due to incorrectly placed braces, wrong variable used in an expression, or other hard to see effects. To consider such mistakes seriously the programmer needs confidence in the tool. (This is compounded by the natural human tendency when criticized to first think of reasons why the criticizer is wrong.)
Thirdly, if programmers do not ignore the complaints they merely try to eliminate the symptom rather than look for the cause. For example, a complaint about an uninitialized variable may be resolved by simply initializing the variable in its declaration, even if the programmer believes that during actual execution the variable will be always initialized. This is counter-productive because in our experience "uninitialized variable" is often not a symptom of lack of initialization, but something quite different, for example, misspelled variable in some condition. The programmer needs to search for the real cause of the symptom, and he will not do that unless he has the confidence that the symptom can cause actual failure during execution.

Another important aspect is the incorporation of project-specific information. The ultimate goal of this effort is a tool that is sufficiently powerful and sufficiently flexible that any restrictions can be checked mechanically. Examples of restrictions are interfaces to lower levels of software, any protocols to be observed, or just the rules of the programming language. We target the tool towards checking the constraints of the lower levels of software, as opposed to checking the functionality to be provided to the higher levels, not because most errors lie in violations of interfaces. The reason is that many errors can manifest themselves either as violations of functionality, or violations of lower level interfaces; detecting them as the latter is much cheaper mainly because the programmer need not provide any detailed functional specifications. Moreover,

```
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

the correct use of interfaces implies reproducibility, which is a prerequisite of many functional validation strategies.

The tool described here is most closely related to the falsifier of [14], which is based on symbolic execution. Our tool also uses symbolic execution, but overcomes its main weakness, path explosion, by using data-flow analysis to find potential faults.

The paper does not describe any details of the algorithms [15], but instead reports which issues we found important, how we resolved them, and what was the experience with the solution. Section 2 describes what kind of problems our tool looks for. Section 3 explains how it decides whether a particular control-flow path can be followed during actual execution. Sections 4, 5, 6, 7 describe the design of our tool. Sections 8 reports on our experience with it. Finally Sections 9 discusses how our design decisions turned out in retrospect.

2 Symptoms

Most literature on software reliability ties the notion of a "fault" to a "failure", namely discrepancy between user expectations and observed behavior during execution. Since source-code analysis does not involve any execution, we will not use the term "failure". In defining "fault" we will follow [16], where an intended program is postulated; a "fault" is then any discrepancy between the intended and the actual program. Instead of the term "failure" we will use the term "fault symptom", an example of which is the use of an uninitialized variable. In general, every source-code analysis tool offers a list of symptoms it can detect and a user selects from this list symptoms to be reported because in his intended program none of the selected symptoms should occur.

Throughout the paper we will refer to the example of Figure 1. (We will use the C language for all examples.) The right hand side contains an intended program, the left hand side has an error in the statement `if(!I)`, which should be `if(I)`. As a result the function on the left may return an uninitialized value of `X`, which is a symptom that can be detected.

Our tool currently detects about 80 symptoms, which are divided into three levels of difficulty. These symptoms are both generic (i.e., violations of the programming language semantics), as well as project specific (mainly violations of constraints on data base accesses).

Difficulty 0 symptoms are those commonly detected by lint-like tools. They include unused variables, mismatches in printf arguments, different initializations of one external variable in two files, portability problems, and others. Project specific symptoms may involve naming conventions, or restrictions on the invocation of certain functions.

Symptoms of difficulty 0 are distinguished from the others by the fact that they are not associated with a path. They are typically associated with just one statement, and they could cause a failure no matter which path is followed in reaching that statement. These kinds of symptoms are not the focus of our effort, because many of them are covered by existing commercial tools. We detect only those difficulty 0 symptoms that have been explicitly requested by a user. They are easy to add, but the falsifier does not try to make certain that they will necessarily cause the software to fail. For a user, a false symptom of difficulty 0 is more acceptable than a false symptom of difficulty 1 or 2, because it is easier to decide whether a symptom of difficulty 0 could actually cause a failure.

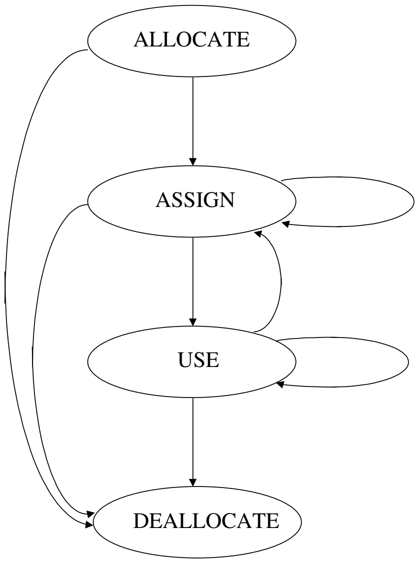

Figure 2: Legal operations on a variable

Difficulty 1 symptoms are violations of a finite state property along some execution path. There are many symptoms expressible in this form; for example, consider Figure 2, showing possible operations on a variable. All the allowed operations are shown by arrows. Disallowed operations include:
Uninitialized variable corresponds to going from ALLOCATE directly to USE.
Use of deallocated memory corresponds to going from DEALLOCATE to ASSIGN, USE or DEALLOCATE.
Memory leak corresponds to avoiding DEALLOCATE.
These are all examples of difficulty 1 symptoms. Project specific symptoms of difficulty 1 are typically some protocol violations, because they can be also expressed by finite state properties.

Difficulty 2 symptoms include “index out of array range”, “dereferencing a null pointer”, “failed assertion”. In general all symptoms of difficulty 2 are expressible as assertions, which is what we rely on for project specific symptoms of difficulty 2. Such assertions are inserted automatically into the code during parsing, and therefore do not require special consideration.

Symptoms of difficulty 2 are more difficult to detect than difficulty 1 for the following reason. Both share the problem of identifying a feasible path whose execution will cause the symptom. However, suppose that we ignore the problem of determining whether a path is actually feasible, and are merely given a single path and asked whether a particular symptom occurs along that path. If we were asked whether a given variable is uninitialized along that path, we can answer that question easily without needing arithmetic. But to determine whether an index is out of range we need to perform some theorem proving, which is time consuming.

3 Feasibility

A falsifier reports a fault symptom only if it is associated with a feasible path. To illustrate the issue of feasible paths consider Figure 1. The program contains two if-statements, each allowing flow in two directions, giving in total four topological paths. However, only two of the paths are feasible in the sense that it is possible for execution to follow them. An example of an infeasible path in Figure 1(a) is the path following the then-clauses of both if-statements.

Checking feasibility is expensive (see Section 7) and therefore compilers and lint tools do not do it. Thus they cannot distinguish between the incorrect program of Figure 1(a) and the correct program of Figure 1(b). How they handle this uncertainty differentiates compiler from lint. Compilers tend to report a variable as uninitialized only if it is uninitialized along all paths. Lint tools would report a variable as uninitialized if it is uninitialized along any path, feasible or not.

In general, feasibility is to be determined for a given partial path in a given procedure. It involves two more issues besides checking the consistency of conditions along the path. First, the conditions along the path may depend on input parameters, whose possible values are set outside the given procedure. Secondly, the conditions along the path may depend on variables assigned as a side-effects of procedure calls, whose bodies may or may not be available to the falsifier.

Both of these issue are related and are resolved similarly. First of all, inter-procedural analysis (Section 5) calculates side-effects and can derive some information about input parameters and variables assigned as a result of procedure calls. But such information is not always sufficiently accurate to be used as evidence that a failure is possible. Therefore as in [6] we also rely on conditions appearing explicitly in the program.

For example, consider the question of whether Figure 3 contains the symptom of a fault. (The function would fail if it were ever called with a null argument.) If the falsifier can find a call to `P` with a null argument then it will certainly report the symptom. In the absence of such a call nothing will be reported. However, for the program of Figure 4 the if-statement `if(A)` is considered evidence that `A` could be null, and therefore the unconditional dereferencing of `A` is considered the symptom of a fault.

In general, the testing of a condition by a program is considered as evidence that the condition can be sometimes true and sometimes false. There are two arguments for this. First, the author of the procedure as in Figure 4 apparently expected that it could be called with a null argument. Secondly, anybody else examining the implementation of such a function would see the test of `A` and might conclude that the procedure P may be called with a null.

This approach to determining legal input values is not ideal for two reasons. First, it is very conservative – if a programmer completely forgot the possibility that an argument could be null, the falsifier will see no evidence that it could be null, and therefore will fail to report any fault. On the other hand, an unnecessary

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

Figure 3: Will fail whenever A is null

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

  return *A;
}
```

Figure 4: The testing of A is evidence that it could be sometimes null

test may cause the falsifier to assume that an input value is legal, although the code may actually rely on never being given such a value. This could lead to a false error report; however, there is a general agreement that even this false error indicates an inconsistency in the program.

4 Representation

The falsifier does its work in four stages – parsing, building of a graph representation, inter-procedural analysis, and intra-procedural analysis.

For parsing we use the C++ compiler of [17]. It provides a parse tree for each procedure as well as type analysis of all variables.

The parse tree of each procedure is then translated into a control and data flow graph described below. In addition, a call graph is built, which records which procedures call which other procedures. During this graph-building stage symptoms of difficulty 0 are also reported.

After all the files have been parsed and all the graphs saved on disk, inter-procedural analysis calculates procedure side-effects, and then intra-procedural analysis looks for symptoms of difficulty 1 and 2.

The whole process is incremental in the following sense. All the graphs representing all the procedures are stored on disk. When a program file is updated, only that file needs to be parsed again. Inter- and intra-procedural analysis needs to be redone for all the procedures in the changed file. Also it needs to be redone for a procedure calling any procedure whose side-effect information changed.

We will not describe the details of the control and data flow graphs, but mention only those aspects that will be needed later. One important property is that nothing is abstracted away; all the details of the program as well as all the details of C++ semantics are recorded in the graphs, because any detail can make a difference in deciding feasibility of a path.

The control flow graph is built in the standard way. Nodes representing executable statements are connected if control can flow from one to the other. If the flow is only conditional then the condition is attached to the arc connecting the two nodes. For example, an if-statement generates a fork and a merge in the control flow graph. The conditions of the if-statement are attached to the edges of the fork as well as the edges of the merge. Please note that the fork and the merge may have different conditions in case of gotos out of the if-statement. The reason for placing the conditions at the merge is that all our analysis is backwards – from a potential site of a fault symptom we search for a feasible path. We need the conditions at the merge nodes in order to detect infeasibility as soon as possible.

The data flow graph, representing expressions, is connected directly into the control flow graph. For example, a node on the control flow graph representing an assignment has one input edge from the data flow graph representing the value assigned.

An important activity of the falsifier is simplification of the graph representation. The simplification has two goals – reducing the graph in size for efficiency, but more importantly to make the graph canonical where possible. To reduce the graph size we perform some standard compiler optimizations [18], such as constant propagation, value numbering, etc. We perform no code motion because that would make it harder to report a fault symptom to the user in term of his program.

For ease of deduction it is important to make the graph representation as canonical as possible, that is, expressions need to be rewritten to allow maximum sharing of subexpressions. For example, suppose that the two expressions `2*A-2*B < 11` and `B-A+6 < 1` appear in the program. They will be brought into the formats `A-B < 6` and `A-B > 5`, sharing the common subexpression `A-B` so that if their consistency ever needs to be established the result will be immediate.

5 Inter-procedural analysis

In general, source-code analysis cannot handle an entire software system as a whole, because it would be looking for feasible paths with lengths measured in billions. (Currently we can find feasible paths involving only hundreds of statements.) Therefore source-code analysis tools look for a fault symptom in one procedure at a time. On the other hand, a procedure cannot be analyzed in complete isolation for two reasons – first we need to know possible values of input parameters of the procedure being analyzed, and secondly we need to know the side-effects of other procedures being called. Inter-procedural analysis provides both of these kinds of information.

We use a standard approach to the problem [19, 20, 21]. It consists of calculating side-effects of individual procedures and propagating those side-effects to their callers. A typical example of a side-effect is an assignment to a global variable. Inter-procedural analysis gives information of three types – side-effects that always happen, side-effects that can never happen, and side-effects that can happen conditionally. Only the first two types are usable for error reporting. Side-effects that can happen only conditionally are not usable because inter-procedural analysis relies on execution-insensitive data flow analysis (see Section 6), which may think that something is possible, although it is not in reality.

We will not discuss any details of inter-procedural analysis because it is currently being redone. We found the existing implementation inadequate for several reasons.
First the execution-insensitive nature of the analysis forces us to be very conservative causing many errors to go unreported.
Secondly, traditional inter-procedural alias analysis has cubic worst case performance, which is unaccept-

able.
Thirdly, recursive data-structures are handled by “k-limiting”, which allows a trade-off between performance and accuracy; we find that values of k with acceptable performance are too inaccurate.
Currently we are addressing these issues.

Besides inter-procedural analysis we also obtain side-effect information by inlining called procedures into the procedure being analyzed. This is more accurate because that allows us to consider the exact conditions of side-effects. Inlining is particularly important for the analysis of loops, which are represented as recursive procedures; by default each loop is unrolled (i.e., expanded inline) one or two times. Besides loops we also expand inline all procedures shorter than a given length.

6 Intra-procedural analysis

Intra-procedural analysis looks for symptoms of difficulty 1 and 2 inside a given procedure. If such a symptom is found the user is given a symbolic execution trace starting with the procedure entry and terminated with the symptom of the fault. This trace may contain calls to other procedures; if the call has been expanded inline, then the trace includes execution of the called procedure, otherwise the trace relies on the side-effects calculated by inter-procedural analysis.

It is up to the user to decide manually what errors exist along that trace. Determining the cause of the fault is often non-trivial, although it tends to be easier than with actual execution traces obtained through testing. We do provide some hints as to what parts of the trace are likely to be relevant to the fault, but those hints are only heuristic and cannot be completely relied on. (Later in this section we will describe how the hints are generated.) In general, the user does need help in locating the cause of the fault, but we do not see a possibility that it could be completely automated.

For the purpose of finding a feasible path leading to a fault symptom we use two types of analysis – data flow analysis [3, 18, 22] and symbolic execution [23, 24]. Both propagate information along paths in a program. They differ in the way they deal with reconverging paths. Symbolic execution considers all feasible paths, the number of which could grow exponentially with program size. (In the presence of loops, the number of paths would be infinite, but we replace loops by recursive procedures.) In contrast, data-flow analysis combines information calculated for two reconverging paths, resulting in behavior that is linear in program size.

For example, consider Figure 1, and assume that it is to be analyzed by a certain type of data flow analysis (execution-insensitive – see below). We could proceed in a forward fashion from procedure entry to exit. Before reaching the statement `if(!I)` two partial paths have been analyzed – one corresponding to the case that `A` is true, and the other that `A` is false. Along one path `X` is assigned 0, along the other `X` is not asigned anything. Data-flow analysis merges these two pieces of information into one, namely “At the statement `if(!I)`, `X` may or may not be initialized.” Whereas, symbolic execution does not merge any information, and keeps all the details of both paths.

Data-flow analysis has the advantage of efficiency, but its results cannot be used to report a fault symptom. Any time information is merged, some details are lost, and we cannot be sure that there is actually a feasible path to the symptom. The results of symbolic execution can be used to report a fault symptom, but symbolic execution suffers from an exponential explosion of paths.

Therefore we combine the advantages of the two types of analyses. Data-flow analysis is used as a filter to screen out areas definitely containing no fault symptom. If it does find the possibility of a fault,

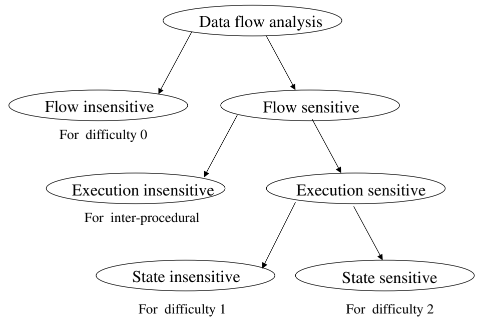

Figure 5: Categories of data flow analysis

execution-sensitive analysis (see below) calculates a "bundle" of paths that can lead to the symptom of the fault. Symbolic execution is then restricted to this bundle of paths, which is normally small enough to make it efficient. Symbolic execution selects one path from the bundle to be reported to the user; however, the bundle information is also given to the user because knowing which other paths lead to the symptoms helps to determine what is relevant to the fault.

Data-flow analysis is divided into four categories (see Figure 5) with increasing power and run time. The least powerful analysis is **flow-insensitive**, which treats a given program merely as a set of statements ignoring how control flows among them. For example, in Figure 1 flow-insensitive analysis would find that `X` is both assigned and used somewhere in the procedure, indicating no problem. We use flow-insensitive analysis for some problems of difficulty 0, for example, variables that are never assigned, or parameters that are never used. However, this type of analysis is rarely useful because the types of symptoms it tends to detect are detected even by compilers.

Flow-sensitive analysis treats the program as an directed graph, where nodes representing statements are connected if control can flow from one to the other. Therefore flow-sensitive is more precise than flow-insensitive. We divide it into execution-sensitive and execution-insensitive.

Execution-insensitive analysis ignores all conditions in if-statements, case-statements, etc, and assumes that all paths are feasible. Such analysis would detect that `X` may be uninitialized in both Figure 1(a) and Figure 1(b); it cannot distinguish between them. Flow-sensitive, execution-insensitive analysis is the workhorse of compilers and lint tools. We use it during inter-procedural analysis and also as a "pre-filter" for memory leaks and related symptoms.

Execution-sensitive analysis [22, 25, 15] does consider conditions along a path in determining its feasibility. We distinguish between state-sensitive and state-insensitive analysis, and use state-sensitive analysis for symptoms of difficulty 2, while state-insensitive analysis for symptoms of difficulty 1.

State-insensitive analysis [22] determines the feasibility of a path by checking consistency of the conditions along the path, but ignores how the variables appearing in the conditions are manipulated by the program. For example, in Figure 1 state-insensitive analysis would consider the two conditions $A = 0 \wedge I \neq 0$ perfectly consistent, although they will never be true during actual execution. The reason is that it looks at the conditions in isolation from how the variables are manipulated. Therefore state-insensitive analysis cannot distinguish between Figure 1(a) and Figure 1(b). It could distinguish between the two if the statements `if(!I)` and `if(I)` were changed to `if(!A)` and `if(A)` respectively.

State-sensitive analysis [15] does not ignore anything in determining path feasibility. Therefore it would distinguish between Figure 1(a) and Figure 1(b). Of course, even state-sensitive analysis is merely data flow analysis, and therefore can serve only as a filter in deciding whether to use symbolic execution. The inaccuracy of state-sensitive analysis comes from the fact that merging information along reconverging paths effectively removes some conditions from consideration; but the consistency of the remaining conditions is determined exactly.

One major difference between execution-sensitive and execution-insensitive analysis lies in the number of pieces of information derived. Execution-insensitive analysis always derives just one piece of information for all the paths in a program – for example, "is a given variable ever assigned?". It calculates just one piece of information because at path reconvergence points it always merges information collected for individual paths.

In contrast, execution-sensitive analysis does not always merge; at some path reconvergence points it may keep both pieces of information, not unlike symbolic execution. For each piece of information it also remembers the conditions characterizing the bundle of paths for which the information was calculated. For example, in Figure 1(a) an execution-sensitive analysis would calculate three bundles.
The first bundle is characterized by the condition $I \neq 0$, which contains two topological paths, and for this bundle the variable `X` is not used.
The second bundle is characterized by the conditions $I = 0 \wedge A \neq 0$, which contains one topological path, and for this bundle the variable `X` is used and initialized.
The third bundle is characterized by the conditions $I = 0 \wedge A = 0$, which contains one topological path, and for this bundle the variable `X` is used and but not initialized.
Since an error can happen only in the third bundle, symbolic execution need to consider only the one path in that bundle.

7 Theorem prover

The purpose of the theorem prover is to determine whether a particular set of conditions is consistent. A falsifier needs to prove simpler theorems than a verifier [8, 13, 26] for the following reason. In a falsifier the burden of proof is on the tool to show that a path containing a fault is feasible. Therefore if a particular path is too difficult to decide, it is acceptable to give up and report nothing. However, in a verifier the burden of proof is on the user to show the absence of error. If the verifier is unable to prove that the evidence provided by the user is sufficient, the user must be able to provide some hints or more evidence. The evidence tends to be in the form of assertions describing the state of the program, or in the form of properties of some procedures, on which the program relies. Such assertions need to describe the results of iteration; to describe the results of iteration requires quantifiers or some other forms of iteration. Therefore the theorem prover of a verifier must deal with quantifiers or some form of induction. But for a falsifier, no quantifiers are needed, or more exactly, all variables have an implicit existential quantifier.

The input into the theorem prover is a set of conditions and the output is one of three: "the conditions are satisfiable", "the conditions are not satisfiable", "cannot decide within given time limit". Satisfiable conditions normally imply that we found a feasible path containing a fault. If the prover cannot decide then no report is generated for the user.

We use two levels of the theorem provers – state-sensitive and state-insensitive. The state-insensitive is used by state-insensitive data-flow analysis; the more accurate state-sensitive prover is used by state-sensitive data-flow analysis and symbolic execution. The difference lies in the treatment of the variables appearing in the list of conditions. While the state-insensitive prover assumes that all the variables are independent of each other, the state-sensitive prover takes into consideration the graph structure defining the values of the variables. This then provides the difference between the state-sensitive and state-insensitive analysis as explained in Section 6.

The state-insensitive prover works simply by anding all the conditions, while ignoring the graph. If that results in an inconsistency then the conditions are inconsistent even in the more accurate state-sensitive sense and the candidate fault is definitely not feasible. However, conditions consistent in state-insensitive way might not be in state-sensitive way.

The state-sensitive prover works by building a set of equalities and inequalities concerning edges in the graph. The equalities and inequalities are obtained by propagating information about inputs of a node to its outputs or vice versa. For example, from $a \in [0, 3], b \in [1, 4]$ we can deduce $a + b \in [1, 7]$.

The rules for propagating information correspond to the laws of arithmetic, or any other data domain. More rules make the prover more powerful, but also slower; therefore it is important to have no more rules than necessary. We have given to the prover only those rules that we actually found needed in our application domain of design automation software. It turned out that it was sufficient to have only the rules concerning the operations and relations $+, -, <, =, \neq$; this includes pointer arithmetic. For multiplication we needed to understand multiplication by a constant plus the rule $a \times b = 0$ iff $a = 0$ or $b = 0$. The theorem prover does not need to understand all the other operators of arithmetic, bitwise operators, etc; the simplifications performed in the course of building the graphs of Section 4 are sufficient. In addition to the surprisingly small needs for arithmetic we also needed very little propositional calculus sophistication – resolution with unit clauses proved sufficient.

On the other hand, it is very important for the prover to understand the interaction between the data flow and control flow. The issue concerns variables appearing in the input to the prover. These variables refer to values they had been assigned in the program, and the state-sensitive prover needs to use these values. One approach to the issue, represented by [24], considers one path at a time; then all the variables can be replaced unambiguously by their values. However, considering one path at a time is too inefficient; it is necessary to consider a bundle of paths at a time. In that case a variable could be assigned different values along different paths in the bundle. Some approaches to this problem, e.g., [27] give the prover a complete description of what values each variable is assigned under what conditions. In our experience we found such a description unacceptably large.

Our approach to this problem is related to [8], where the prover manipulates program segments. Our prover simplifies the given graph under the conditions given to it. That may resolve the values of variables; and even if it does not, it will identify two variables as having the same contents if that is implied by the given conditions.

Both the state-sensitive and state-insensitive provers are incremental in the sense that adding a new condition does not require recalculating what was derived for the prior conditions. This is important because the analysis of Section 6 traverses paths and constantly asks the prover whether the partial path traversed so far is feasible.

8 Experience

This section reports how many fault symptoms the falsifier found in software in our application area, which is design automation software. This type of software manipulates representations of VLSI chips stored in a special purpose data base. The data base is accessed via procedural interface. In addition to accessing the data base the software does a lot of dynamic memory and pointer manipulations. There are some floating-point calculations, but most arithmetic is integer.

Table 1 lists four programs and for each it gives its size, time to process each stage and number of fault symptoms found. All four programs have been through unit and system testing, and have been used for some time inside the corporation. The program labeled 42K does placement of circuits, the programs labeled 274K and 485K are two logic synthesis systems, and 46K is the falsifier 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.

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

Table 1: Fault symptoms per program

Symptom	Occurrences	Difficulty	Application specific
Data base access without proper permission	1269	0	YES
Incorrect use of message catalogue	1202	0	YES
Use of uninitialized variable	844	1	NO
64 bit portability problem	705	0	NO
Passing NULL to a function	198	2	SOME
Dereferencing NULL pointer	131	2	NO
Memory leak	123	1	SOME
Constant expression	93	0	SOME
Failure to cleanup after data base access	83	1	YES
Error in printf	51	0	NO
Side effect in an assert statement	51	0	SOME
A certain incorrect data base access	32	0	YES
Unreachable statement	28	0	SOME
Index out of array range	23	2	SOME
Stametent without any effect	21	0	NO
Freed memory passed out of a function	18	1	SOME
Inconsistent declarations of extern variables	13	0	NO

Table 2: Fault symptoms by frequency

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 the four phases –
“parsing”: running the parser of [17],
“graph”: building and simplifying the graphs, as well as reporting symptoms of difficulty 0 (Section 4)
“inter-proc”: inter-procedural analysis (Section 5),
“intra-proc: searching for symptoms of difficulty 1 and 2 (Section 6)

In the program labeled 46K there were no fault symptoms found during this statistics collecting run, because the falsifier is used on itself regularly. The programs with sizes 274K and 485K also had been analyzed in the past; the symptoms reported this time reflect code changes, new feature of the falsifier, and faults not corrected in the past. The large number of difficulty 0 symptoms for 274K include some violations of project specific coding rules, not all of which would necessarily make the software fail. Other than that all the other reported symptoms have been inspected manually to make sure that they are real faults; they do not represent merely bad programming practices.

Table 2 lists some commonly occurring fault symptoms. These have been collected over a longer period of time, not just from the four runs of Table 1. For each symptom we report how many times it occurred in various programs, the symptom's difficulty, and whether it is application specific. Application specific symptoms typically involve some restrictions imposed by the data bases storing the VLSI circuits. Some symptoms are partially application specific; for example, to discover some memory leaks it is necessary to know which library functions allocate memory and which free it.

From the results of Table 1 we see that our tool takes between 5 and 10 minutes per thousand lines of code. It averages on the order of one fault per thousand lines of "tested" code. From Table 2 we can see the importance of application specific information, especially for symptoms of difficulty 0.

9 Conclusions and Discussion

The paper presented the design of a software falsifier and experience with its use. In contrast with lint and verifiers, falsifier's goal is to report only those faults that can actually cause the software to fail. One consequence of this conservatism is that it may not report all faults. On the other hand, the conservatism allows to be target faults of higher difficulty.

Some of the decisions made in the design of the falsifier turned out to be right, and some did not. We will describe them and report on our experience with those design decisions.

The decision to avoid false error even at the expense of missing real errors proved correct. Our users are very sensitive to the "noise" generated by many lint tools and find such tools unacceptable. At the present time our tool may still report some false errors due to inadequate calculation of procedure side-effect, a shortcoming being worked on.

The decision to sacrifice speed in order to avoid false errors proved acceptable. Of course, performance is important, but it is not essential to run at compiler speed. As a result users do not use the falsifier as often as a compiler, but only after each major update.

The method of determining legal inputs (Section 3) proved acceptable, but not ideal for the reasons discussed at the end of Section 3. However, we are aware of no alternative requiring no extra information from the programmer.

Our graph representation (Section 4) proved reasonable, but too inefficient in storage. Both disk space and memory consumption turned out to be more critical than originally anticipated. As a result the falsifier is limited to software systems of about a million lines of code. (Larger pieces need to be broken into sub-projects in order for the call graph to fit into memory.) Also the intra-procedural analysis cannot handle procedures exceeding approximately 10,000 lines of code. Neither limitation is a problem in our application area.

The main shortcoming of the tool is its weak inter-procedural analysis (Section 5) and the inlining of small procedures at the place of invocation. The inlining is unavoidable if we want to consider feasible paths only. However, inlining should not be done statically before knowing what fault symptoms we will be looking for. Different amount of inlining needs to be done for different symptoms.

The decision to represent loops as recursive procedures (Section 4) turned out very controversial. Some people object to it on the grounds that we can locate faults that involve only two loop iterations. In comparison, compilers and lint tools do data flow analysis on loops and thus can consider arbitrary number of iteration. That is not a valid criticism because a falsifier cannot rely on execution-insensitive data flow analysis as compilers do, and execution-sensitive analysis does not in general converge for loops. The real disadvantage of representing loops as recursive procedure is that it prevents us from handling backward gotos. In addition, our treatment of loops forces us to assume algol-like variable scoping rather than the simpler C-like variable scoping. The main advantage of handling loops as recursive procedures is that the two constructs share the same issues, which can be handled just once. However, it is not clear that this advantage outweighs the two disadvantages.

The interaction between data-flow analysis and symbolic execution (Section 6) is essential to the success of the falsifier. The algorithms for execution-sensitive data flow analysis are the main contribution of this effort.

The integration of the theorem prover with the graph representation (Section 7) also proved very important. The theorem prover needs very little sophistication in terms of arithmetic or propositional calculus, but needs detailed understanding of the program structure.

Overall, our results so far point to three conclusions. First, programmers do want the functionality of a falsifier. Secondly, although calculation of feasible paths is expensive it can be done sufficiently efficiently. Thirdly, the conservatism of a falsifier allows us to target and discover faults that are not found by other methods.

10 Acknowledgments

The author is grateful to R. Puri, L. Stok, D. Kung, 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] M. Fagan, "Design and code inspection to reduce errors in program development", *IBM system journal*, vol. 15, pp. 182–211, 1976.

[2] D. Freedman and G. Weinberger., *Handbook of Walkthroughs, Inspections and Technical Reviews*, Little Brown, Boston, MA, USA, 1990.

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

[4] I. F. Darwin, *Checking C programs with lint*, O'Reilly, 1988.

[5] 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.

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

[7] F. Belli and Radu Crisan, "Towards automation of checklist-based code reviews", *in Proceedings of the 7th International Symposium on Software Reliability Engineering*, pp. 24–33. IEEE, October 1996.

[8] R. S. Boyer and J S. Moore, *A Computational Logic*, Academic Press, London, England, 1979.

[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] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill, "Symbolic model checking for sequential circuit verification", *IEEE Transactions on Computer-Aided Design*, vol. 13, pp. 401–424, April 1994.

[12] J. V. Guttag and J. J. Horning, *Larch: Languages and Tools for Formal Specification*, Springer-Verlag, 1993.

[13] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas, "A turorial introduction to pvs", *in Workshop on Industrial Strength Formal Specification Techniques*, April 1995.

[14] *Prefix Corporation*, http://www.prefixco.com/, 1999.

[15] D. Brand, "Error detection by data flow analysis restricted to executable paths", Technical Report RC-21484, IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY 10598, May 1999.

[16] J. D. Musa, A. Iannino, and K. Okumoto, *Software Reliability: Measurement, Prediction, Application*, McGraw-Hill Book Company, 1987.

[17] 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.

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

[19] M. Sharir and A. Pnueli, "Two approaches to interprocedural data flow analysis", in S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pp. 189–234. Prentice-Hall, 1981.

[20] K. D. Cooper and K. Kennedy, "Interprocedural side-effect analysis in linear time", *in Proceedings of ACM Conference on Programming Language Design and Implementation*, pp. 57–66. ACM, 1988.

[21] M. Burke, "An interval-based approach to exhaustive and incremental interprocedural data flow analysis", *ACM Transactions on Programming Languages and Systems*, vol. 12, July 1990.

[22] 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.

[23] S. Galand and G. Loncour, "Structured implementation of symbolic execution: a first part in a program verifier", *Information Processing Letters*, vol. 3, pp. 97–103, March 1975.

[24] S. I. Hantler and J. C. King, "Introduction to proving the correctness of programs", *ACM Computing Surveys*, vol. 8, pp. 331–353, September 1976.

[25] 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.

[26] J. von Wright, “Representing higher-order logic proofs in hol”, *The Computer Journal*, vol. 38, pp. 171–179, July 1995.

[27] 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.