RC22905 (W0304-120) April 11, 2003
Computer Science

IBM Research Report

Arithmetic Reasoning for Static Analysis of Software

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

Florian Krohm
IBM Microelectronics
East Fishkill, NY

Research Division
Almaden - Austin - Beijing - Haifa - India - 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 , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home.

Arithmetic Reasoning for Static Analysis of Software

Abstract

Software and hardware verification has shown the ability to prove the correctness of many sophisticated algorithms and to discover many subtle errors. But the verification tools are not in every-day use by non-specialists. This paper describes several of the verification methods in a tool, which is in every-day use by programmers. The paper concentrates on theorem proving and related aspects.

1 Introduction

Software is an increasingly larger component of computer systems, which makes it both an increasingly larger contributor and impediment to overall system reliability. Therefore there is an increased urgency to bringing into everyday practice the verification methods developed over the past 40 years. This paper describes several such methods in practical use.

The most widely practiced approach to increasing reliability is testing. Testing is *dynamic analysis* in the sense that the program under test is executed on concrete inputs. In contrast, *static analysis* does not rely on executing the program, instead it analyzes its source description.

Dynamic methods include the popular Purify [1] and similar tools. Dynamic methods take as input an executable version of a program and a set of test cases. They execute the testcases and compare the results against desired behavior. The desired behavior can be in the form of specific output values, or in the form of *generic requirements* independent of desired functionality, e.g., "no access to freed memory", no "memory leaks", etc.

Static tools include compilers, lint [2, 3, 4, 5], and formal verifiers [6, 7, 8, 9, 10, 11]. Their input is the source of the whole program, or only a portion of it. They issue a complaint if they find a possibility of an input that violates desired behavior. The desired behavior can be in the form of a specific input-output relationship, or the same generic requirements mentioned for dynamic tools.

Static and dynamic tools have their advantages and disadvantages, which makes them complementary. We will talk about a static tool, whose goal is to address a major disadvantage of static tools. Lint-like tools and formal verifiers are famous for their tendency to issue invalid complaints (called "false positives" by some, and "false negatives" by others); namely complaints that do not represent any defect in the given program.

There are three main sources of invalid complaints. Tools that require some form of specification may issue complaints that are due to mistakes in the specification or due to inadequacy of the specification. Tools that require an abstract model of an actual program may report problems in the model that have no counterpart in the actual program. And finally, no tool can decide the correctness of every program and therefore there is always some uncertainty. Tools that aim at verification (guarantee of correctness) issue a complaint even when uncertain whether it represent a real defect.

Our tool (called BEAM) has the goal of maximizing the number of problems detected subject to three constraints:
1) The input is the actual program exactly as given to a compiler.
2) No specifications are required.
3) Every complaint issued should represent a valid defect that the user will want to correct.
These are our goals and in Section 7 we will discuss to what degree we can meet them.

This implies that the tool's goal is not verification, namely forming a proof of correctness and issuing a complaint if not successful. Its goal is falsification, namely forming a proof of incorrectness and issuing a complaint if successful.

In its goals BEAM is closest to the tool Prefix [12], which is based on symbolic execution [13]. BEAM also uses symbolic execution, but only as a confirmation of the results of other analyses, which avoid the path explosion problem of symbolic execution. Some methods of our tool are also related to Meta-Level Compilation [14]. However, that is not a falsifier – it does not avoid invalid complaints due to infeasible paths. There is more work related to algorithms described in this paper and that will be discussed in the pertinent sections.

The choice of falsification rather than verification is key to acceptability by users. It has also several implications on algorithms used. For example, induction

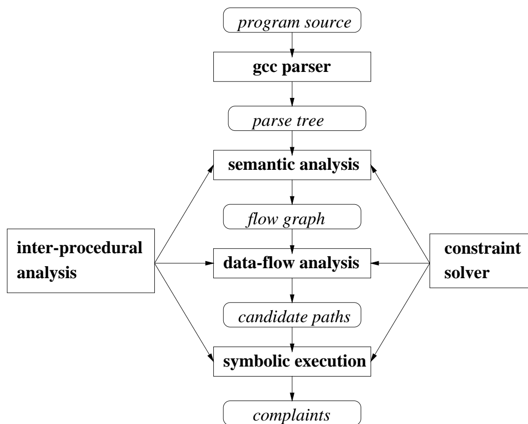

Figure 1: Structure of BEAM

plays a very different role than in verification. Further, it is not sufficient to report an error whenever there are input values that would cause it; the tool must also provide evidence that the program is intended to work on such values.

The above considerations are applicable to both hardware and software. In fact, since a function can be compiled into either software or hardware, the distinction loses its relevance. However, for the purposes of static analysis there is one relevant distinction, namely the way computer operations (e.g. addition) are modeled. The issue is that computer operations violate the rules of arithmetic when overflow occurs. If overflow is considered a normal occurrence in the design then it is appropriate to use a finite model of computer arithmetic. On the other hand, if overflow is considered an abnormal occurrence then an infinite model of arithmetic is more appropriate; that is, arithmetic is performed on unbounded numbers, which are truncated when stored into memory. The decision depends on how the designer thinks – if he thinks in terms of bit operations then the analysis tool should do the same. But if he assumes all the rules of arithmetic when reasoning about his design, then the tool must use the same rules in its reasoning. In addition, the tool needs to detect where the rules could be violated and whether it could cause a problem. Our tool is geared to software analysis in the sense that it uses the infinite model of arithmetic.

The paper is organized as follows. We start with a general overview of the tool and gradually focus towards one portion – a solver of arithmetic constraints.

2 Overview

This section uses an an example to illustrate the operation of our tool and in particular to illustrate the use of the constraint solver.

```
#include <stdlib.h>

void foo(int N, int I, char X)
{
  int J;
  char *A, *B;
  L1: A = (char *) calloc(N, 1);
  L2: B = (char *) calloc(N+5, 1);
  L3: B[N-1] = X;
  L4: if (B[I])
  L5:   J = I;
      else
  L6:   J = I-1;
  L7: A[J] = 1;
}
```

Figure 2: Sample program

Fig. 1 shows the structure of the tool. Its input is a source program as shown in Fig. 2. There are several defects in this program, such as memory leaks and possible indices out of array bounds. We will concentrate on the last statement L7 and will consider the question of whether the index J could exceed the bound of the array A.

The source program is input into the gcc parser, which we have modified to produce a parse tree plus other information. This extra information is necessary to relate any complaints to the source program, in terms understandable to the user. That is in general an extremely difficult problem, but it is not the subject of this paper.

From the parse tree we generate a control and data flow graph as shown in Fig. 3. It is a rather standard flow graph with the possible exception of the mux node, which we will explain later. The flow graph is a simplification of our actual representation, and shows only those aspects needed to explain how our constraint solver operates on the graph. Each control flow node has a label C1, ..., C9, which is attached for the purposes of this explanation.

The first node C1 represents the action of `calloc(N, 1)`. It allocates a piece of memory, which is not named in the program, but we will call it U. There is more information associated with the memory location U, namely its size and the fact that it is initialized to all zero.

The second node C2 assigns a pointer to U into the memory location A. And similarly for nodes C3 and C4.

The assignmemt statement C5 illustrates a com-

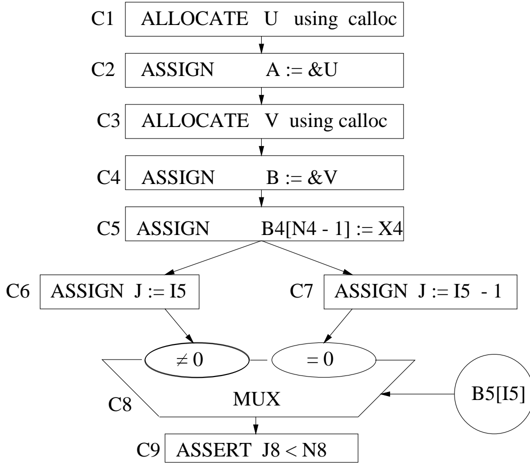

Figure 3: Original flow graph

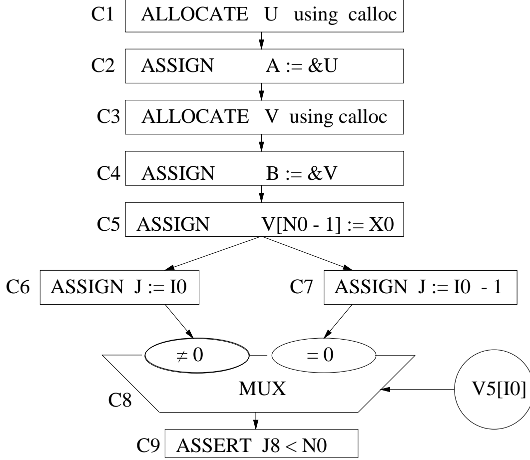

Figure 4: Reduced flow graph

mon notation used for exposition. If a location name (e.g., A, B, U, V, I, J) is followed by a number then it refers to the contents of the memory location. And the number indicates at what point in the control flow graph the contents should be obtained. For example B4 refers to the contents of the location B in the memory state after C4 (which happens to be `&V`).

All the control flow nodes are considered in a functional sense; namely they take as input a state of memory and produce a new state of memory. The if-condition does not represent any updates to memory and therefore no node is shown for it.

The two branches of the if-statement are represented by the nodes C6 and C7. The text "I5-1" is an example of a shorthand for several data flow nodes, which perform the computation and whose output is then input into the node C7.

The nodes C6 and C7 produce different states of memory. One of them is selected by the mux C8, depending on the value of the control input B5[I5]. The control input is compared with the predicates written inside each input pin. Each predicate is a set of integers in which the control value may lie. If it does then the corresponding input is selected to become the value on the output of the mux.

This representation allow us to combine the control and data flow graphs into one flow graph, with every node producing a value. As we will see the constraint solver can then generate conditions on the values of control nodes and treat them in a uniform manner with the data flow nodes. An example of a data flow node is the the node B5[I5] controlling the mux.

For statement L7 of Fig. 2 we show only the node C9 representing the assertion that the index J does not exceed the array bound.

After the flow graph of Fig. 3 is built, it is simplified into that of Fig. 4 using rewrite rules (Section 3). All the components in Fig. 1 below semantic analysis then operate on the graph in search for feasible paths. That process will be explained in later sections.

The example of Fig. 2 does not illustrate our handling of loops. Loops are represented as recursive procedures and their properties are calculated during inter-procedural analysis. These properties describe which locations could be modified, which pointers will be dereferenced, etc.

The data flow analysis that derives the properties is path-insensitive (i.e. does not determine feasibility of paths) and is a form of induction. However, induction cannot be used in a search for a feasible path leading to an error. A successful inductive proof can be used to terminate a search. However, a failing inductive proof cannot be used by a falsifier to report an error, because the failure might be due to inadequacy of an inductive hypothesis, whether generated automatically or manually. We can report an error only if we can exhibit a feasible path leading to it. Therefore the search unrolls loops a fixed amount of times, and searches in that limited space. All path-sensitive analyses thus operates on an acyclic flow graph, and so does the constraint solver.

3 Rewrite Rules

The meaning of the the graph nodes is defined by rewrite rules. Each rewrite rule can have two effects – it can modify the graph, and it can create constraints. Both effects can be conditional.

We illustrate the modification of the graph by a rewrite rule, that understands the ASSIGN nodes and fetches of values from memory. The effect of the rule is shown in Fig. 4. For example, the rule understands

that none of the assignments affect the location I, and therefore I5 = I0. That is, the content of I after C5 is the same as the content of I before C1. And since this is true unconditionally, the graph can be modified replacing I5 with I0. Similarly it understands that the assignment C4 affects the location B in a way that B4 = &V.

In general, the assignment rule takes as input an assignment node (e.g. C5: V[N0-1] := X0) and a memory access (e.g. V5[I0]) and has three possible outcomes. If it determines that the assignment has no effect on the memory access then it generates the equality constraint V5[I0] = V4[I0]. If it determines that the assignment does affect the given memory access then it generates the equality constraint V5[I0] = X0. If it cannot decide then it does nothing.

If a rewrite rule can generate an equality unconditionally then the equality can be used to modify the graph. And that is how the graph is simplified before any analysis starts (Fig. 4). The next section illustrates the situation of conditional rewrite rules.

4 Example of Reasoning

Before explaining path traversal and the constraint solver we show how they work together on the example of Fig. 4. They generate a sequence of constraints as in Fig. 5.

The constraint 1. is the negation of the assertion. If it is found consistent with constraints of some path then we have a feasible path to the error.

We perform backward data-flow analysis and therefore we will illustrate backward path traversal. The path traversal considers both of the paths in Fig. 4 starting with the left branch, which is executed under condition 2.

Imposing the constraint 2. on the node representing V5[I0] triggers rewrite rules that understand semantics of nodes attached to that node V5[I0]. In our case it triggers the rule for mux, which generates constraint 3. It says that the state of memory after C8 is the same as after C6.

All references to C8 are replaced by C6. This replacement makes J8 = J6 (constraint 4.)

The rule of assignment generates 5. because the value of J after C6 is I0 (independently of any assumptions).

After substituting all the equalities we get 6. From that the rule of assignment can conclude that the assignment C5 does not effect V5[I0], resulting in constraint 7.

The assignment C4 has no effect on V, which is expressed by constraint 8.

The rule that understands calloc() knows that its result is all zero, concluding with 9.

```
1. J8 >= N0          (Condition violating assert)
2. V5[I0] != 0       (Chose left branch)
3. C8 = C6           (From 2. by rule of mux)
4. J8 = J6           (From 3. by equality)
5. J6 = I0           (By rule of assignment)
6. I0 >= N0          (From 1, 4, 5 by equality)
7. V5[I0] = V4[I0]   (From 6. by rule of assignment)
8. V4[I0] = V3[I0]   (By rule of assignment)
9. V3[I0] = 0        (By rule of calloc)
10. 0 != 0           (From 2, 7, 8, 9 by equality)
```

Figure 5: Constraints generated for V5[I0] != 0

```
1. J8 >= N0       (Condition violating assert)
11. V5[I0] == 0   (Chose right branch)
12. C8 = C7       (From 11 by  rule of mux)
13. J8 = J7       (From 12 by equality)
14. J7 = I0-1     (By rule of assignment)
```

Figure 6: Constraints generated for V5[I0] == 0

However, after substituting all the equalities into 2. we get 10. which is a contradiction. It indicates that whenever execution follows the left branch the index in statement L7 will not exceed the array bound.

It is important to note that path traversal merely chose to follow the left branch between C6 and C8. From that the constraint solver followed the consequences all the way to node C3. Putting the programming language semantics into the constraint solver, rather than the path traversal is important in reducing the number paths considered.

In searching for a feasible path, path traversal backs up to the mux C8 to chose the right branch. All the deductions in Fig. 5 are on a stack, and backing up to C8 pops the stack leaving only the entry 1. After that the stack grows again with constraints of the right branch, see Fig. 6. This time no contradiction is derived implying a feasible path along which the index J will exceed the bound of A.

Should this be reported to the user? A verifier would report it because no preconditions were specified that would prevent the index from being out of bounds. In verification, the burden of proof is on the user to show the impossibility of error.

In falsification the burden of proof is on the tool to show that the procedure `foo(N,I,X)` was intended to function even if give inputs `I > N`. A falsifier must find evidence of intentions without requiring any specifications. We recognize two forms of such evidence. First, if there were a call to `foo(N,I,X)` with `I > N`,

then the problem could be reported. A second form of evidence would be an explicit test, say `if (I <= N)`, inside the function foo(). Such a test indicates that the programmer does expect parameters that make the test evaluate sometimes to true and sometimes to false.

In general, a problem can be reported only after finding a path that is not only consistent with the condition of the error, but actually implies it. This is not the case in Fig. 2, therefore our tool could not report any possibility of index J out of array range.

5 Path Traversal

Path traversal is needed to find a feasible path leading to an error. The main issue in path traversal is to avoid enumerating all the paths. As mentioned above, all loops and recursive procedures have been unrolled a fixed number of times, so that the number of paths is actually finite. However, it is in general exponential in the size of the program. We avoid path enumeration by a combination of path-sensitive data-flow analysis and symbolic execution. They perform path traversal in different ways, but both call the constraint solver to determine feasibility of paths. The main purpose of this section is to explain how path traversal and constraint solving interact.

There are two basic approaches to the interaction between path traversal and constraint solving. Verification condition generators (first implemented by [15]) collect the conditions of a path in a formula and pass it to a stand-alone theorem prover, which then decides the formula's satisfiability. An alternative approach pioneered by [16] lets the the program representation itself be the formula. We use the latter approach because it lends itself to incrementality, which is key to reducing the problem of path explosion.

Our constraint solver is incremental in two ways. First when extending a partial path a new constraint may be given to the solver (e.g., line 2 in Fig. 5). That may trigger calculation of some implications from the new constraint, but does not require reprocessing previous constraints. Secondly when path traversal abandons a particular path it may backtrack to a previous branch point. That causes the constraint solver to undo all implications derived along the abandoned sub-path, while preserving all previous implications that are still valid.

This incrementality is implemented by placing all the solvers information on a stack. For efficiency the stack is implemented as a hash table.

6 Constraint Solver

As illustrated above the constraint solver accepts constraints, and determines whether they are consistent. A constraint is any relation operator of the programming language between two nodes of the flow graph (e.g. $A < B$). In general, any such predicate can be considered as a function returning 0 or 1. And that is indeed how we treat all predicates with the exception of $=, \neq$ and $<, \leq, >, \geq$ on integers.

The solver is a combination of rewrite rules (expressing the semantics of the programming language) plus special handling of linear integer arithmetic. The latter is needed because integer arithmetic is key for efficient reasoning about programs. And the rewrite rules provide flexibility in handling a variety of programming language constructs.

6.1 Integer Arithmetic

Linear Integer arithmetic is the subset of integer arithmetic containing the operations $+, -, <$ in addition to equality. It is of great interest for two reasons. First it is indispensable to reasoning about programs, and secondly it is decidable [17] (in contrast to general arithmetic containing multiplication).

However, the decision procedure has a super exponential upper and lower bound [18, 19]. Therefore the key to a practical solver is to find a subset of arithmetic rules as small as possible for efficiency, yet sufficient for the purposes of the application. There are several such subsets [20, 21, 22]. This section presents our subset of rules which is smaller than existing ones.

In our case of a falsifier, all formulas are existentially quantified, which matches the formulation of integer programming [23]. Integer programming can decide the satisfiability of a number of constraints of the form
$a_1x_1 + ... + a_nx_n < c$
where $a_1, ..., a_n, c$ are constants and $x_1, ..., x_n$ are variables.

We extend the formulation to be of the form
$a_1x_1 + ... + a_nx_n \in R$
where R is any set of integers. This extension allows more efficient handling of constraints of the form
$a_1x_1 + ... + a_nx_n \neq c$
which can be represented as
$a_1x_1 + ... + a_nx_n \in (-\infty, c) \cup (c, \infty)$.
This extension causes us to sacrifice the convexity of the solution space, which makes linear programming approaches inapplicable.

Secondly we restrict $n \leq 2$. That is, we consider constraints of the form
$A \in R$ or
$a_1A_1 + a_2A_2 \in R$
where A, A_1, A_2 are nodes in the flow graph. Constraints of this form are very common in program analysis, while cases of $n > 2$ are much less common. Moreover, the case of $n \leq 2$ can be handled more effi-

ciently than $n > 2$.

More general constraints with $n > 2$ can still be represented because any of the nodes A_1, A_2 can be the results of arbitrary operations. However, reasoning about more general constraints is done through rewrite rules in the flow graph as opposed to any special reasoning about inequalities.

As a result of our approach to arithmetic there are three types of constraints
1) equalities (e.g., $A = B$)
2) unary constraints (e.g. $A < 5$)
3) binary constraints (e.g. $A < B$)

At any point in time the constraint solver maintains a list of constraints with all their implications. Any addition of a new constraint triggers derivation of implications, and application of rewrite rules. The objective is to keep reducing the ranges R in the unary and binary constraints. If we can deduce a constraint with the empty range R then we proved that the given set of constraints is not satisfiable; that commonly implies the impossibility of a particular error.

We always try to keep the flow graph and the constraints in a canonical representation, wherever possible. For that purpose we consider a fixed topological ordering of all the nodes in the flow graph. In this ordering first come nodes representing constants, then nodes representing memory locations, then nodes representing input value, and then come all the other nodes in topological order. This topological order determines the ordering of terms in binary constraints and the ordering of inputs to commutative graph nodes.

6.1.1 Equality

We represent equality as an equivalence relation using the union/find algorithm of [24]. This representation allows average O(log n) time to retrieve the representative of any node. Equality constraints are imposed simply by replacing every reference to a node by its representative.

The axiom of substitutivity $A = B \Rightarrow f(A) = f(B)$ is implemented using a rewrite rule, common called "common term elimination" or "value numbering". We made no attempt to implement full congruence closure.

6.1.2 Unary Constraints

Unary constraints are of the form $A \in R$, where A is a node in the flow graph and R is a set of integers. They are represented by storing a range of integers (denoted $\Re(A)$) with every node A. Initially $\Re(A)$ is the maximal range determined from the corresponding declaration in the program source.

Unary constraints are imposed by range propagation on the flow graph. For example, consider a node $A = B + C$. A change of $\Re(B)$ will trigger a possible reduction of $\Re(A)$ and $\Re(C)$ so that
$\Re(A) \subseteq \Re(B) + \Re(C)$,
$\Re(C) \subseteq \Re(A) - \Re(B)$.

In the above we rely on the usual extension of all the arithmetic operators to sets. For example,
$R_1 + R_2 = \{x | x = r_1 + r_2, r_1 \in R_1, r_2 \in R_2\}$.

In addition, the following rule relates unary constraint and equality constraints:
$A \in \{k\} \Leftrightarrow A = K$
That means that if the range of A consists of the single number k then A equals the node K representing the number k.

6.1.3 Binary Constraints

Binary constraints are of the form $aA + bB \in R$, where A, B are nodes in the flow graph, a, b are integer coefficients and R is a set of integers.

Binary constraints are stored in an array, and are normalized so that A comes before B in the topological order, $a > 0$, $b \neq 0$, (a, b) are relatively prime.

The following operation is applied as a generalization of transitivity $(x < y \wedge y < z \Rightarrow x < z - 1)$. Suppose there is a pair of binary constraints
$aA + b_1 B \in R_1$,
$cC + b_2 B \in R_2$,
sharing a node B (A and C may or may not be identical). Then we generate another binary constraint
$b_2 aA - b_1 cC \in b_2 R_1 - b_1 R_2$, obtained by subtracting the original two so as to eliminate B.

The following rules relate binary constraints to unary constraints. That means, any of the ranges below may be reduced to as to ensure these relations.
$\Re(aA + bB) \subseteq a\Re(A) + b\Re(B)$
$\Re(A) \subseteq (\Re(aA + bB) - \Re(B))/a$

Finally the following rule relates binary constraints to equality constraints
$A - B \in \{0\} \Rightarrow A = B$.

6.1.4 Soundness and Completeness

The constraint solver is sound in the sense that it derives a constraint with the empty range only if the given set of constraints is unsatisfiable. However, it may fail to discover some unsatisfiable sets of constraints, for example, $A \neq B, B \neq C, C \neq A, A \in \{0, 1\}, B \in \{0, 1\}, C \in \{0, 1\}$.

Failure to discover an inconsistency would lead to reporting an error that can happen only along an in-

Table 1: Frequency of problems reported

NUM	COMPLAINT
1107	**Dereference of NULL**
448	**Uninitialized return value**
317	**Test without effect**
236	**Uninitialized variable**
195	Missing parentheses from macro definition
143	Error in printf arguments
118	**Missing break in a switch statement**
71	**Subscript out of array range**
38	Statement without effect
33	Unused assignment
31	Mistake in operator precedence
24	**Memory leak**
23	Mistake in parameters to strncmp
19	**Wrong deallocating function**
16	**Failure to close a file**
11	**Function returning dangling pointer**
11	Incorrectly nested loops
10	Incorrect use of vararg
8	**Access to freed memory**
8	Unintended semicolon
7	**Loop without effect**
7	Unintended assignment in an if-condition
2	**Memory freed twice**
2	**Locks set without releasing**
1	Problem of portability to 64 bit machines

feasible path. While the solver could be extended towards completeness, it proved unnecessary. Doing so would slow it down, reducing the number of valid complaints reported.

7 Experience

Our tool has been used over a year inside IBM to analyze C and C++ programs, mainly in firmware and system software development. The purpose of this section is to give the reader an idea of its effectiveness, what problems it detects most frequently, and to what degree the tool meets the goals in Section 1.

During the week March 1 - 7, 2003 there were 8,247 source files analyzed through BEAM, containing 4,111,775 lines. Many of the runs involved the same file being reanalyzed after some modification. Therefore it is more meaningful to provide statistics only for the first run of each file. There were 1,794 unique source files containing 1,577,015 lines. For those 2,943 complaints were issued in the categories shown in Table 1. Bold font indicates complaints requiring the algorithms of this paper. The other complaints are purely syntactic analysis, where feasibility of paths need not be determined.

In Section 1 we stated three goals.

Goal 1) requires that we process the source code in the form given to the compiler. We can do that for close to 100% of the functions. For the rest there is a problem of language compliance. Our various users have various compilers running on various operating systems. Each compiler offers language extensions, which would be rejected by other compilers. We make an effort to configure our gcc-based parser to handle such extensions, but that effort is not always 100% successful.

Goal 2) states that no specifications be required. We honor this goal, but we allow users to give us additional information about their application; for example, which functions allocate memory, which functions terminate a thread, etc. Also application specific coding rules can be checked for; for example after some operation certain cleanup is required, or the results of a function need to be checked for -1 before used as an index. For user acceptability it is essential that this extra information be considered part of the tool and that the responsibility for its correctness lie on the people supporting the tool, not the programmer.

Goal 3) states that we report only problems indicating a defect in the source code. We do not have statistics on how well we meet this goal, but apparently to sufficient degree for user acceptance. There are two major causes of invalid complaints. One cause is insufficient information about user's function library. For example, if the tool does not know that a particular function reports an internal error, then it may tell the user that of a crash that will happen after he called that function – not a useful piece of information. (This problem is eliminated for users who provide information about their function library.) The second cause of invalid complaints is the fact that a tool can report only symptoms of problems, and may misread user's intentions. For example, sometimes programmers intentionally force a crash. A more common example of invalid complaints are those reporting unnecessary computation, for example, statements without effect. There is a trade-off between reporting invalid complaints and missing valid defects. That trade-off is different for different source files, and it is up-to the users to configure the tool to their needs. It is essential that the tool allows the elimination of invalid complaints to any degree desired.

8 Conclusions

We have learned three main lessons. First, the most important for acceptability is meeting the goals of accepting the program source exactly as is and avoiding

invalid complaints. Inability to turn off invalid complaints would cause users to ignore all complaints.

Secondly, the main complexity issue is avoiding path enumeration. Our approach through a combination of data-flow analysis, symbolic execution and incremental constraint solver proved successful, although efficiency remains a problem.

Thirdly, it turned out that logical reasoning in software is extremely shallow, and a fast and simple constraint solver is sufficient.

There are several areas where our tool needs improvements. Better inter-procedural analysis would allow us to report more sophisticated errors. (We measure "sophistication" by the amount of code that has to be examined in determining the cause of a problem.) Efficiency improvements are needed in both the constraint solver and path traversal. For better usability we need to be able to explain the chain of reasoning so that users can more easily find the cause of a problem. In general, as we drive towards more sophisticated problems, the user will need to spend more time for each complaint issued, which calls for improving user interaction and for avoiding invalid complaints.

References

[1] Rational, *http://www.rational.com/*.

[2] S. C. Johnson, "Lint, a C program checker," Tech. Rep. 65, Bell Laboratories, Murray Hill, NJ 07974, 1978.

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

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

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

[7] S. M. German, "Automating proofs of the absence of common runtime errors," in *Proceedings of the 5th ACM Symposium on Principles of Programming Languages*, pp. 105–118, ACM, 1978.

[8] R. B. Jones, D. L. Dill, and J. R. Burch, "Efficient validity checking for processor verification," in *Proceedings of the IEEE International Conference on Computer-Aided Design*, (Santa Clara, CA), pp. 2–6, IEEE, November 1995.

[9] P. J. Windley, "Formal modeling and verification of microprocessors," *IEEE Transactions on Computers*, vol. 44, January 1995.

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

[11] B. Jacobs, J. van den Berg, M. Huisman, and M. van Berkum, "Reasoning about java classes," *SIGPLAN Notices*, vol. 33, pp. 329–340, October 1998.

[12] W. R. Bush, J. D. Pincus, and D. J. Sielaff, "A static analyzer for finding dynamic programming errors," *Software Practice and experience*, vol. 30, no. 7, pp. 775–802, 2000.

[13] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[14] D. R. Engler, D. Y. Chen, and A. Chou, "Bugs as inconsistent behavior: A general approach to inferring errors in systems code," in *Symposium on Operating Systems Principles*, pp. 57–72, 2001.

[15] J. C. King, *A Program Verifier*. PhD thesis, Carnegie-Mellon University, Pittsburg, Pennsylvania, September 1969.

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

[17] M. Presburger, "On the completeness of certain system of arithmetic in which addition occurs as the only operation," *History and Philosophy of Logic*, vol. 12, no. 2, pp. 225–233, 1991.

[18] M. J. Fischer and M. D. Rabin, "Supper-exponential complexity of presburger arithmetic," Tech. Rep. 43, MIT, 1974.

[19] D. C. Oppen, "A 2**2**2**pn upper bound on the complexity of presburger arithmetic," *Journal of Computer and System Sciences*, vol. 16, no. 3, pp. 323–332, 1978.

[20] R. E. Shostak, "A practical decision procedure for arithmetic," *Journal of ACM*, vol. 26, pp. 351–360, April 1979.

[21] G. Nelson and D. C. Oppen., "Simplification by cooperating decision procedures," *ACM Transactions on Programming Languages and Systems*, vol. 1, October 1979.

[22] C. Barrett, D. L. Dill, and J. Levitt, "Validity checking for combinations of theories with equality," in *First International conference on formal methods in computer-aided design*, (Aachen, Germany), pp. 187–201, Lecture Notes in Computer Science, November 1996.

[23] G. L. Nemhauser and L. A. Wosley, *Integer and Combinatorial Optimization*. John Wiley and Sons, 1988.

[24] R. E. Tarjan, "Efficiency of a good but not linear set union algorithm," *Journal of the ACM*, vol. 22, no. 2, pp. 215–225, 1975.