Propositional Combination of CBI Predicates

Piramanayagam Arumuga Nainar
Computer Sciences Department
University of Wisconsin-Madison
⟨arumuga@cs.wisc.edu⟩

Ting Chen
Computer Science Department
University of Wisconsin-Madison
⟨tchen@cs.wisc.edu⟩

Jake Rosin
Computer Science Department
University of Wisconsin-Madison
⟨rosin@cs.wisc.edu⟩

Abstract

Cooperative Bug Isolation (CBI) is a technique to find bugs in programs, that analyzes data collected from program executions. At each program point CBI identifies boolean expressions, called predicates, to be instrumented. We augment CBI's bug predictive ability by combining these predicates using logical operators (conjunction and disjunction). The motivation is that a complex predicate will provide more information to the programmer by narrowing down possible program states. We present both qualitative and quantitative evidence that complex predicates are useful. We discuss a new metric that uses program structure to quantify the usefulness of complex predicates. Using this metric, we could eliminate a large number of spurious complex predicates from consideration. Finally we discuss the effect of sparse random sampling on the usefulness of complex predicates.

1. Introduction

The Cooperative Bug Isolation (CBI) Project [8] finds bugs in programs by analyzing reports collected from software executing in the hands of end users. To use CBI, the software must be compiled with an instrumenting compiler that inserts snippets that evaluate boolean expressions (called predicates) at various program points. Predicates are designed to capture program behaviors such as results of function calls, directions of branches or values of variables. At the end of each execution, the instrumented program generates a report that contains the number of times each predicate was measured, and the number of times each was found to be true. Statistical debugging ([7] and [11]) is used to analyze these reports and find predicates that are predictive of failure. These predicates are then ranked and presented to the developer.

CBI gathers execution reports by using valuable CPU cycles at end user machines. It is essential to make those cycles worthwhile by extracting every bit of useful information from them. However the current statistical analysis algorithms ([7] and [11]) consider predicates in isolation from one another. They overlook potentially useful relations between predicates. Predicates are expressions involving program variables at different program points and hence may be related by control and data dependences. We propose to capture these relations by building *complex* predicates from the set of currently instrumented predicates (which we refer to as simple predicates). Since predicates are boolean expressions, they are combined using logical operators (such as *and* and *or*). We construct complex predicates and include them in the input to the statistical analysis algorithms.

There are two approaches to combine predicates using logical operators:

1. Explicitly monitor the complex predicate by changing the output of the instrumenting compiler.
2. Estimate the value of the complex predicate from the values of its components.

The first approach will yield a precise value but needs significant modifications to existing infrastructure. The second approach will be less precise (as described later) but requires only few modifications to existing infrastructure (and none to the instrumenting compiler). In this project, we implement the second approach, that will serve as a proof of concept for complex predicates, as well as a justification for a future attempt at incorporating them into the compiler.

The remainder of this report is organized as follows. Section 2 introduces CBI and explains the motivation for complex predicates. Section 3 gives a precise definition of complex predicates and discusses the trade-offs in our implementation. Section 4 describes two metrics to evaluate the usefulness of a complex predicate. Section 5 describes two case studies that demonstrate the usefulness of complex predicates.

Section 6 presents the results of experiments conducted on a large suite of test programs. Section 6.4 discusses the effect of sparse random sampling on complex predicates. Sparse random sampling is a technique used by CBI to reduce the runtime overhead on the instrumented programs. Section 7 discusses some related work and section 8 concludes.

2. Background

CBI uses lightweight instrumentation to collect feedback reports that contain truth values of predicates in an execution as well as the outcome[1] of the execution. A large number of these reports are collected and analyzed using statistical debugging techniques. The analysis described in [7] computes a numeric score corresponding to each predicate. The score is called *Importance* and is computed as follows:

The truth values of a predicate P from all the runs can be aggregated into four values:

1. $S(P \text{ observed})$ and $F(P \text{ observed})$, the number of successful and failed runs respectively, in which the value of P was evaluated.
2. $S(P)$ and $F(P)$, the number of successful and failed runs respectively, in which the value of P was evaluated and was found to be *true*.

Using these values, two scores of bug relevance are calculated. They are:

1. $F(P)$. A good predictor must predict a large number of failed runs.
2. $Increase(P)$, the amount by which P being true increases the probability of failure over simply reaching the line where P is defined. It is computed as follows:

[1] crash and non-crash or some other binary classification

$$Increase(P) \equiv \frac{F(P)}{S(P)+F(P)} - \frac{F(P \text{ observed})}{S(P \text{ observed}) + F(P \text{ observed})} \quad (1)$$

Both of these scores are independent but good dimensions of a bug predictor. These dimensions are combined into a single value by taking their harmonic mean. Since $Increase(P)$ is bounded by 1, the $F(P)$ component in $Importance$ is normalized over the total number of failed runs $NumF$ after a logarithmic transformation. The overall metric is:

$$Importance(P) = \frac{2}{\frac{1}{Increase(P)} + \frac{1}{log(F(P))/log(NumF)}} \quad (2)$$

To eliminate different predicates that predict the same bug, only the top ranked predictor is presented to the user. To handle the case where multiple bugs are present, all the failed runs in which the top predicate is *true* are eliminated and the remaining predicates are ranked by recomputing their scores in the remaining set of runs. This process of eliminating runs continues until there are no remaining failed runs or no remaining predicates.

The output of the analysis will contain the list of predicates that had the highest score during any iteration of the redundancy elimination algorithm. This list can be used by the programmer to track down bugs, or as input to other automated analysis tools.

Complex predicates can improve the analysis described above in three ways: high scoring predicates, better bug predictors and as a better complement to automated tools.

Predicates with High Scores: A statistical analysis with a larger input set will find more predicates with high scores. Moreover, complex predicates can track deep properties about program behavior. So they are more likely to differentiate faulty runs from successful ones and thus have higher scores. Predicates with high scores are useful because they capture a program behavior that is highly correlated with failure. Even if they do not directly point at the faulty location, they can help identify use cases or modules in which the failure occurs. Though a high scoring predicate that has no relation to the bug can be a hindrance to the user, a programmer with sufficient knowledge about the software can easily filter out such predicates.

Better Bug Predictors: A predicate obtained by combining simple predicates using logical operators can yield better bug predictors. These predicates can improve the analysis of *super* and *sub* bug predictors. A *super* bug predictor is one that is true in a large number of failures but also true in a large number of successful runs. In other words, it is not a *specific* enough predictor of failure. A possible (but not the only) cause for *super* predictors is a non-deterministic bug that does not necessarily cause a failure when it is triggered. A *super* predictor is complete (i.e. predicts all failures) but it is not sound (predicts some successes also). False positives could be eliminated by taking a conjunction of this predictor with another predicate that captures another aspect of the failure.

A *sub* bug predictor is one that is true in a small number of failed executions. In other words, it is not *sensitive* enough to predict this failure. A *sub* bug predictor is sound (predicts only failures) but is not complete (contains false negatives). It means that this predicate is not the only cause of failures. False negatives could be eliminated by taking a disjunction with another predictor. It is important to note that in software with multiple bugs, the analysis may find a disjunction of predictors of individual bugs as a predictor for the whole set of failures. The user should keep this in mind while using a disjunction predictor to track down a bug.

A $perfect$ bug predictor is one that is true in all failed executions and false in all successful ones - in other words, it is both sound and complete. Such predictors indicate a bug which is completely deterministic given the involved predicates. Generally speaking the closer a predictor is to $perfect$ the more useful it is to a programmer. By eliminating false positives and negatives conjunctions and disjunctions can move $super$ and sub predictors closer to $perfect$ ones.

Automated Tools: Unlike programmers, automated tools are scalable and can benefit from more data points. BTRACE [6] is one such tool that finds the shortest feasible path in the program that visits a given set of predicates. Since complex predicates increase the input to BTRACE, BTRACE can making precise decisions at branch-merge points, where previously it did not have enough information to choose the right branch direction to include in its output.

3. Complex Predicates

This section gives a precise definition of complex predicates and discusses the trade-offs in our implementation.

A complex predicate C is defined as $C = \phi(p_1, p_2, \dots p_k)$ where $p_1, p_2, \dots p_k$ are simple predicates and ϕ is a function that can be computed using only the logical operators *and* and *or*. The operator *not* is not required because any propositional formula can be written in conjunctive normal form (CNF) in which the *not* operator appears only before the literals. By design, the negation of every simple predicate P is also a predicate.

For a predicate P and a run R, $R(P) = 1$ if P was observed to be true at least once during run R. Similarly we could define $R(C)$[2] as follows:

DEFINITION 3.1. *For a complex predicate* $C = \phi(p_1, p_2, \dots p_k)$, $R(C) = 1$ *iff at some point during the execution of the program, C was observed to be true.*

The difficulty with this notion of complex predicates is that C must be explicitly monitored during the program execution. For example, if $C_1 = p_1 \wedge p_2$ then $R(p_1) = 1$ and $R(p_2) = 1$ does not imply that $R(C) = 1$. p_1 and p_2 may be true at different stages of execution but never true at the same time. However, as discussed earlier, explicitly monitoring C requires significant changes to existing infrastructure. In order to be able to estimate the value of C from its components, we adapt a less precise definition as follows:

DEFINITION 3.2. *For a complex predicate* $C = \phi(p_1, p_2, \dots p_k)$, $R(C) = 1$ *iff* $\phi(R(p_1), R(p_2), \dots R(p_k)) = true$

In other words, we assume that R is distributive over ϕ. This can lead to false positives, because $R(C)$ may be computed to 1 when it is actually 0, but no false negatives. The impact of this assumption on the score of C may be either positive or negative depending on whether R failed or succeeded.

There are 2^{2^N} boolean functions of N predicates [10]. This is a prohibitively large number considering that there can be hundreds of simple predicates. To reduce the complexity, we consider only functions of two predicates. There are $2^{2^2} = 16$ such functions and their arguments can be chosen from N predicates in $C_2^N = \frac{N(N-1)}{2}$ ways. Out of the 16 boolean functions of two variables, we consider only conjunction (*and*) and disjunction (*or*) since other functions are more complex and cannot be used effectively by the programmer. Other functions can be easily included in our implementation once their truth tables (discussed in subsection 3.1) are derived correctly. To summarize, we evaluate only $2 \times C_2^N = N(N-1)$ complex predicates. If R is the set of runs being analyzed then the time

[2] For sake of clarity, we use 1, 0 and $true, false$ interchangeably for the values of $R(C)$ and $R(P)$.

complexity required to build complex predicates is $|R|N(N-1) = O(|R|N^2)$

3.1 Three valued logic

This section explains how conjunctions and disjunctions are actually computed. Three-valued logic is used because the value of a predicate in a run may not be conclusive. This can arise in two situations:

1. The predicate was not observed in a run because the run did not reach the line where it was defined.
2. The program reached the line where the predicate was defined but was not observed because of sampling.

In such a case, the value of a predicate P is considered as *unknown*. For the analysis introduced in section 2, it is enough to consider whether $R(P)$ was *true* or *not true* (both $false$ and *unknown*). But while computing complex predicates, the two sub cases of the *not true* case must be considered separately.

Consider a complex predicate $C = p_1 \wedge p_2$. If either p_1 or p_2 was observed $false$ then $R(C) = false$. If both p_1 and p_2 were observed *true*, then C was observed to be true. Otherwise, the value of $R(C)$ is *unknown*. This is shown using a three-valued truth table in Table 1.

Similarly, for a complex predicate $D = p_1 \vee p_2$, if either p_1 or p_2 was observed *true* then $R(D) = true$. If both p_1 and p_2 were observed $false$, then D was observed to be false. Otherwise, the value of $R(D)$ is *unknown*. This is shown using a three-valued truth table in Table 2.

3.2 Interesting Complex Predicate

Even after imposing many constraints, the number of complex predicates is still quadratic in the number of simple predicates. A large number of complex predicates formed by this procedure are likely to be useless in the analysis of the program. A complex predicate that has a lower score than its components is useless. The component (simple) predicate with a higher score is a better predictor of failure, and so the complex predicate adds nothing to the analysis.

DEFINITION 3.3. *A complex predicate* $C = \phi(p_1, p_2, \ldots p_k)$ *is "interesting" iff* $Importance(C) > Importance(p_i)$ *for* $i \in \{1, 2, \ldots k\}$

In the case where the complex predicate has the same score as the component predictor with higher score, the simpler one is preferable. Keeping only interesting combinations of predicates reduces the memory burden of storing them, and helps ensure the utility of a complex predicate that is presented to the user. Definition 3.3 is for the general case and as explained earlier, we explore only the case where $k = 2$ and $\phi \in \{\vee, \wedge\}$.

3.3 Pruning

Forming a complex predicate from its components is a nontrivial task, requiring a conjunction or disjunction for each program run. After this computation is complete the score of the newly formed predicate can be calculated, potentially labeling it uninteresting. In such a case the effort to form the predicate has been wasted. This provides the motivation to prune combinations early based on an estimate of their resulting scores. An upper bound for a predicate's score can be determined by maximizing $F(P)$ and *Increase* under constraints based on the propositional operation. The score of the predicate (Eqn. 2), being a harmonic mean of these two terms, will likewise be maximized.

In this context, the disjunction $D = p_1 \vee p_2$ can be considered as the union of the set of runs where p_1 was *true* and the set where p_2 was *true*. The size of the resulting set is maximized when the two do not overlap, and that is the assumption made when calculating an upper bound on $F(D)$. $S(D)$ is minimized by making the opposite assumption - that one is a subset of the other. The size of the union is thus the size of the superset. The best disjunction that can be formed using p_1 and p_2 will have the following parameters:

$$\begin{aligned}
F(D)' &= F(p_1) + F(p_2) \\
S(D)' &= \max(S(p_1), S(p_2)) \\
F(D\,observed)' &= 0 \\
S(D\,observed)' &= S(D)'
\end{aligned}$$

The second term in *Increase* (Eqn. 1) can only reduce the result, and so it is ignored in calculating an upper bound. For completeness, $S(D \text{ observed})'$ is assigned the same value as $S(D)'$ even though it does not affect *Increase*(D). The $F(D)$ component of $Importance$ is maximized while maximizing *Increase*. Thus the $Importance$ score calculated with the above values is an upper bound on the score of the disjunction of p_1 and p_2.

The conjunction $C = p_1 \wedge p_2$ can be regarded as the intersection of the set of runs where p_1 was *true* and the set where p_2 was *true*. Maximizing $F(C)$ requires that one of the intersecting sets is a subset of the other, making the size of the intersection the size of the smaller set. A minimal $S(C)$ is found when the component sets are non overlapping; the intersection of two such sets is empty.

$$\begin{aligned}
F(C)' &= \min(F(p_1), F(p_2)) \\
S(C)' &= 0
\end{aligned}$$

If the second term in *Increase* is ignored, as with disjunctions, the upper bound of a conjoined predicate's *Increase* score is 1. This is the maximum *Increase* possible, reducing the likelihood of a conjoined predicate being pruned to almost zero. The second term is therefore used for conjunctions to drop the upper bound to a more useful level.

$F(C \text{ observed})$ is minimized by assuming that C was *observed* only in the (failed) runs where it was true. Maximizing $S(C \text{ observed})$ is more difficult. Recall that a conjunction can be considered *observed* if either component is *observed false*, or both are *observed true*. The combination of successful runs where a predicate is *observed false* (union) is maximized if the sets are non overlapping, while the combination where they are *observed true* (intersection) is maximized if one is a subset of the other. By assuming both cases meet that criteria $S(C \text{ observed})$ can be maximized. Its value is the sum of successful runs where p_1 was *observed* and those where p_2 was *observed false*, formed as the difference between $S(p_2 \text{ Observed})$ and $S(p_2)$. Figure 1 depicts the above scenario: notice that the sets $S(\neg p_1)$ and $S(\neg p_2)$ are disjoint and $s(p_1)$ is a subset of $s(p_2)$. Since the result may differ depending on which predicate is chosen as p_1 the larger result is used.

$$\begin{aligned}
F(C\,obs)' &= F(C)' \\
S(C\,obs)' &= \max(S(p_1\,obs) + S(p_2\,obs) - S(p_2), \\
&\qquad S(p_2\,obs) + S(p_1\,obs) - S(p_1))
\end{aligned}$$

Pruning is useful when a computed complex predicate, $C = \phi(p_1, p_2)$, will be thrown away if its score falls below a threshold. Such a threshold can be found in two ways:

1. The scores of p_1 and p_2 that determine whether C is interesting or not.
2. During the redundancy elimination, the highest score among simple predicates can be used as the threshold because if the score is below this value, computing C is useless for this iteration.

Table 1. 3-valued Truth Table for $C = p_1 \wedge p_2$

$P_1 \backslash P_2$	T	F	?
T	T	F	?
F	F	F	F
?	?	F	?

Table 2. 3-valued Truth Table for $D = p_1 \vee p_2$

$P_1 \backslash P_2$	T	F	?
T	T	T	T
F	T	F	?
?	T	?	?

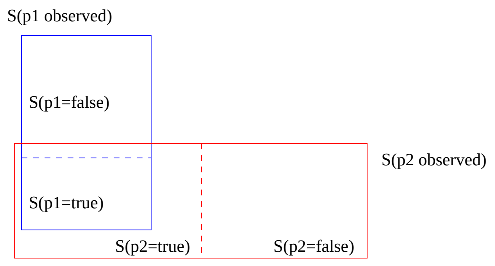

Figure 1. Maximizing S(C observed)

4. Usefulness Metrics for Complex Predicates

While complex predicates could help to better predict bugs, in the worst case, the upper bound on the number of interesting complex predicates is still quadratic in the number of simple predicates. In our experiments, we often observe hundreds of complex predicates with similar or even identical high scores. The overwhelming number of predicates makes it laborious for users to find a complex predicate with which they can start debugging. In this section, we propose two metrics to reduce the number of complex predicates.

The first metric models the debugging effort required from the programmer as a filtering metric. We use the metric defined in [1] for this purpose. In this metric, the score of a predicate is the fraction of code that can be ignored while searching for the bug. We use a similar metric called $effort$ for a complex predicate.

DEFINITION 4.1. *The effort required by a programmer while using a complex predicate $C = \phi(p_1, p_2)$ is inversely proportional to the smaller fraction of code ignored in a breadth-first bidirectional search for p_1 from p_2 and vice-versa.*

The idea behind this metric is that the larger the distance between the two predicates, the greater the effort required. Also, if a large number of other branches are seen during the search, the programmer should keep track of these dependencies too. Like [1], we use the program dependence graph (PDG) to model the program rather than the source code. We perform a BFS starting from p_1 until p_2 is reached and count the total number of vertices visited during the search. The fraction of code covered is the ratio of the number of visited vertices to the total number of PDG vertices.

The second metric is to consider the correlation between the two predicates. Intuitively a complex predicate with two relatively independent predicates is less interesting because it doesn't provide much help to the users in finding anything new, besides the two individual predicates. The correlation between two predicates is defined based on the program dependency graph. Given a single predicate P, we define its *predecessor set* as the set of vertices in the PDG that can influence P.

DEFINITION 4.2. *The correlation between two predicates of a complex predicate is defined as the number of vertices in the intersection of the two predecessor sets.*

The idea behind this metric is that a larger intersection between the *predecessor sets* means it is possible that they are closely related. We expecte correlation to mitigate the issue of disjunctive predicates raised in section 2, namely that the disjunction of predictors for two separate bugs will be scored very highly. The predictors of two unrelated program faults are likely to reside in different areas of the program, and therefore the intersection of their predecessor sets would be smaller than two related predictors for the same bug, which are likely to be in closer proximity.

The above two metrics could be applied both *proactively* and *reactively*. A proactive use of the metrics will prune away complex predicates whose metric values fall below a certain threshold of usefulness. This will eliminate them from being computed and hence improve performance. A reactive use of the metrics will retain all the predicates but break ties by giving higher ranks to those with better values for the metrics. This is desirable if neither computing time nor space is a concern. We use CodeSurfer to build the PDG of a program and to compute these two metrics.

5. Case Studies

This section discusses two cases where complex predicates prove to be useful. The first study is about a memory access bug in Exif 0.6.9, an open source image manipulation program. A complex predicate was useful in increasing the score of an extremely useful bug predictor. The second study uses an input validation bug in `ccrypt` 1.2 to explain how complex predicates can be used to identify *super* bug predictors automatically.

5.1 exif

`exif` 0.6.9 crashes while manipulating a thumbnail in a Canon image. The bug is in function `exif_mnote_data_canon_load` in the module handling Canon images. The following is a snippet from said function:

```
for (i = 0; i < c; i++) {
    ...
    n->count = i + 1;
    ...
    if (o + s > buf_size) return;      (a)
    ...
    n->entries[i].data = malloc(s);    (b)
    ...
}
```

The function skips the call to `malloc` that allocates memory to the pointer `n->entries[i].data` when `o + s > buf_size`. The program crashes when another function `exif_mnote_canon_save` reads from `n->entries[i].data` without checking if the pointer is valid. This is an example of a non-deterministic bug as the program succeeds as long as the uninitialized pointer is not accessed somewhere else.

We generated 1000 runs of the program using randomly generated command line arguments and input images randomly selected from a set of Canon and non-Canon images. There were 934 successful executions and 66 crashes. Applying the redundancy elimination algorithm with only simple predicates produced two predicates that account for all failed runs as shown in Table 3. Studying the source code of the program did not show any obvious relation between the two predictors and the cause of failure. Even though the second predictor is present in the crashing function it was a comparison between two unrelated variables: the loop iterator `i` and the size of the data stored in the traversed array `s`. Also it was *true* in only 31 of the 66 failures.

The analysis had assigned a very low score of 0.0191528 to the predicate P: `o + s > buf_size` despite the fact that it captures the exact source of the uninitialized pointer. Because the bug was non-deterministic, P was also *true* in 335 runs that succeeded. Including complex predicates in the analysis produced one complex predicate shown in Table 4[3]. Conjunction of P with the second predicate P': `offset < len` eliminated all the false positives and thereby earned a very high score. This is an example of how a conjunction can improve the score of a *super* bug predictor. P' is in function `exif_data_load_data` that calls `exif_mnote_data_canon_load` indirectly. It is possible that it captures another condition that drives the bug to cause a crash. If it does, it has to be a deep relationship as we could not find such a relation even after spending a couple of hours trying to understand the source code. However this does not reduce the importance of this result as the conjunction has a very high score compared to P and P'.

A good predictor that could be found using a CBI style analysis is Q: `n->entries[i].data == 0` in function `exif_-mnote_canon_save` or some predicate equivalent to it. Scanning the source file shows that there is no such predicate that is currently instrumented by CBI. If a future instrumentation scheme[4] instruments Q, then even an analysis with simple predicates will find Q as the top bug predictor. However, P is still more useful than Q in identifying the actual source of the *null* pointer. In fact, $P \wedge Q$ will be the perfect result as it captures the bug (skipping the `malloc`) and the trigger (the point where the illegal memory access is performed). Thus, this bug in Exif presents compelling evidence that complex predicates can be better bug predictors.

Threats to validity: There are some internal threats to the validity of the above experiment. Firstly, `exif` 0.6.9 had two other bugs and we had to manually remove command line arguments that trigger those bugs. Secondly, the bug studied here was very rare. In order to get sufficient failed executions, we downscaled the input images by selecting many Canon images (that cause the bug) and some other images (both Canon and non-Canon) that do not trigger the bug. These two changes introduced some bias into the scores of some predicates. For example, our analysis found `remove_thumbnail` in function `main` as a good bug predictor due to the bias introduced by our test suite. However a subjective evaluation of the predicates in Table 3 and Table 4 showed that their scores were not affected by any bias introduced by the test suite. Another threat is that the analysis with complex predicates produced a lot of other predicates with the highest score (0.941385) and we had to scan this list to identify the predicate listed in Table 4. This is not a real threat but is an instance of the numerous complex predicates problem discussed in section 4.

5.2 ccrypt

`ccrypt` 1.2 contains a known bug which can cause a crash on certain user-input - when an `EOF` is entered at the confirmation prompt when overwriting an existing file. Entering `EOF` in other contexts does not cause failure, however, and an examination of the source code can quickly reveal why:

```
/* read a yes/no response from the user */
int prompt(void) {
  ...
  line = xreadline(fin, cmd.name);    (a)
  return (!strcmp(line, "y") ||
     !strcmp(line, "yes"));
}

char *xreadline(FILE *fin, char *myname) {
  ...
  res = fgets(buf, INITSIZE, fin);
  if (res==NULL) {                    (b)
    free(buf);
    return NULL;
  }
  ...
  return buf;
}
```

Calls to `xreadline()`, the function used to get user-input, can return `NULL` under some circumstances. In most cases the value is checked before being dereferenced; in `prompt()` however it is used immediately. `xreadline()` returning `NULL` in `prompt()` should thus be a perfect predictor of failure, occurring in no successful runs and in every failure related to this bug. The branch taken in `xreadline()` is important as well, serving as the moment failure in `prompt()` becomes inevitable. This branch is only taken when the user enters `EOF` on the command line. In mapping the cause of failure, a programmer without a clear understanding of the code is likely to spend time tracking the user-entered `EOF` through `xreadline()` to the `NULL` dereference in `prompt()`, requiring either a visual inspection of the source or use of an interactive debugger. Knowledge of the connection between program events such as these is necessary to make good debugging decisions, e.g. adding a `NULL` check to `prompt()` versus ensuring `xreadline()` always returns a valid pointer. Automated bug analysis should ideally reveal as much of this chain of causation to the programmer as possible.

We generated 1000 runs of `ccrypt`, again using randomly selected command line arguments. Input files included images and text archived from the online documentation of a remote desktop display system. There were 658 successful executions and 342 crashes. All failed runs crashed due to the `NULL` dereference described above - no other bugs were visible to our test suite.

An initial analysis involving only simple predicates found P :`xreadline == 0` as the top predictor of failure: true in no successes and all 342 failed runs, verifying our assumptions. The related predicate Q:`res == (char *)0` scored substantially lower, appearing in all failures but a large number of successes. Q's reported score was low enough that without knowledge of the nature of the bug a programmer would be likely to overlook its significance, and because of its relationship to P it is removed by the redundancy elimination algorithm (see Table 5). More importantly, traditional CBI analysis reveals no connection between the two predictors to the programmer, despite the fact that Q, a necessary but not sufficient condition for failure, is subordinate to P in predicting a crash.

When complex predicates are included in the analysis, a conjunction of P and Q is among the top predictors. This provides little help in finding the bug, which is easily identified by traditional CBI analysis, but it does reveal the nature of Q as a *super* bug predictor.

[3] the second row is the second component of a complex predicate, which is a conjunction as indicated by the keyword *and* at the start

[4] one suggestion is a predicate on scalar parameters to functions

Table 3. Results for Exif with only simple predicates

initial	effective	Predicate	Function	File:line
0.704974	0.704974	new value of len == old value of len	jpeg_data_load_data()	exif-0.6.9/libjpeg/jpeg-data.c:224
0.395001	0.589484	i == s	exif_mnote_data_canon_save()	libexif-0.6.10/libexif/canon/exif-mnote-data-canon.c:176

Table 4. Results for Exif with complex predicates

initial	effective	Predicate	Function	File:line
0.941385	0.941385	o + s > buf_size is TRUE *and* offset < len	exif_mnote_data_canon_load exif_data_load_data	libexif-0.6.10/libexif/canon/exif-mnote-data-canon.c:237 libexif-0.6.10/libexif/exif-data.c:644

Table 5. Results for `ccrypt` with only simple predicates

initial	effective	true successes	false successes	Predicate	Function	File line
0.431678	0.431678	0	342	xreadline == 0	prompt()	src/traverse.c:122
0.385597	0	200	342	res == (char *)0	xreadline()	src/xalloc.c:43

Table 6. Results for `ccrypt` with complex predicates

initial	effective	true successes	false successes	Predicate	Function	File line

0.72814	0	0	342	xreadline == 0
				and res == (char *)0
					prompt()
					xreadline()
						src/traverse.c:12
						src/xalloc.c:43

The conjunction $P \wedge Q$ was observed in more successful runs than P alone, but was true in the same number of successes and failures. That P can be conjoined with Q without affecting P's predictive power demonstrates a connection between the two predicates - in this case suggesting that $P \implies Q$.

This implication was detectable because the experiment was run using complete data collection. Results taken using sparse sampling rates would have made this detection impossible, given the likelihood of Q being unobserved in a run where P was true.

This result provides evidence that complex predicate analysis can automatically group related predicates in ways traditional CBI analysis does not, including the discovery of *super*, *sub* and *perfect* predictor hierarchies and implications. Grouping related predictors statistically provides insight into program structure and execution features which can be used in debugging. This example reiterates that complex predicates can collaborate with tools like BTRACE that produce an execution trace from a set of predicates. Cooperative Bug Isolation can therefore utilize techniques which previously required detailed execution information by generating a facsimile from statistical data.

Threats to validity: The version of `ccrypt` used in this experiment had only one bug visible to our test suite. The statistically demonstrated relationship between P and Q was discovered in the absence of predictors for other bugs, which may have affected the results. Intuitively an unrelated bug would have caused faults in different program runs, allowing the analysis to distinguish between unrelated sets of predictors, but we have not demonstrated this. Further experimentation is needed to determine if this analysis retains this power in the face of multiple bugs. Additionally the predictor $P \wedge Q$, though it scored highly, was not top-ranked, and was in fact discarded by the redundancy elimination algorithm (see Table 6). Knowledge of the code and the component predicates was necessary to distinguish it as important. This once again demonstrates the need for techniques to effectively filter through the large numbers of complex predicates, as discussed in section 4.

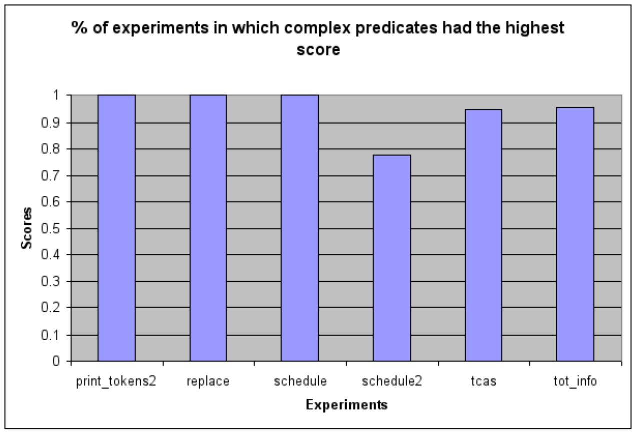

Figure 2. Complex predicates having the highest score

6. Experiments

This section presents quantitative data about the ideas presented in previous sections. This data was collected using the Siemens test suite [5]. There are two configurable parameters for the experiments: the rate of sampling and $effort$ (described in section 4). Unless specified, the default sampling rate is 1 (i.e. complete data collection) and the default $effort$ is 5% (only predicates that are reachable from each other by exploring less than 5% of the program are considered).

6.1 Top Scoring Predicates

Figure 2 plots the percentage of variants within each program for which a complex predicate had the highest score among all predicates. The value is 100% for `print_tokens2, replace` and `schedule` and is close to 100% for the other programs. The results shown in Figure 2 combined with the case studies in section 5 demonstrates the usefulness of complex predicates.

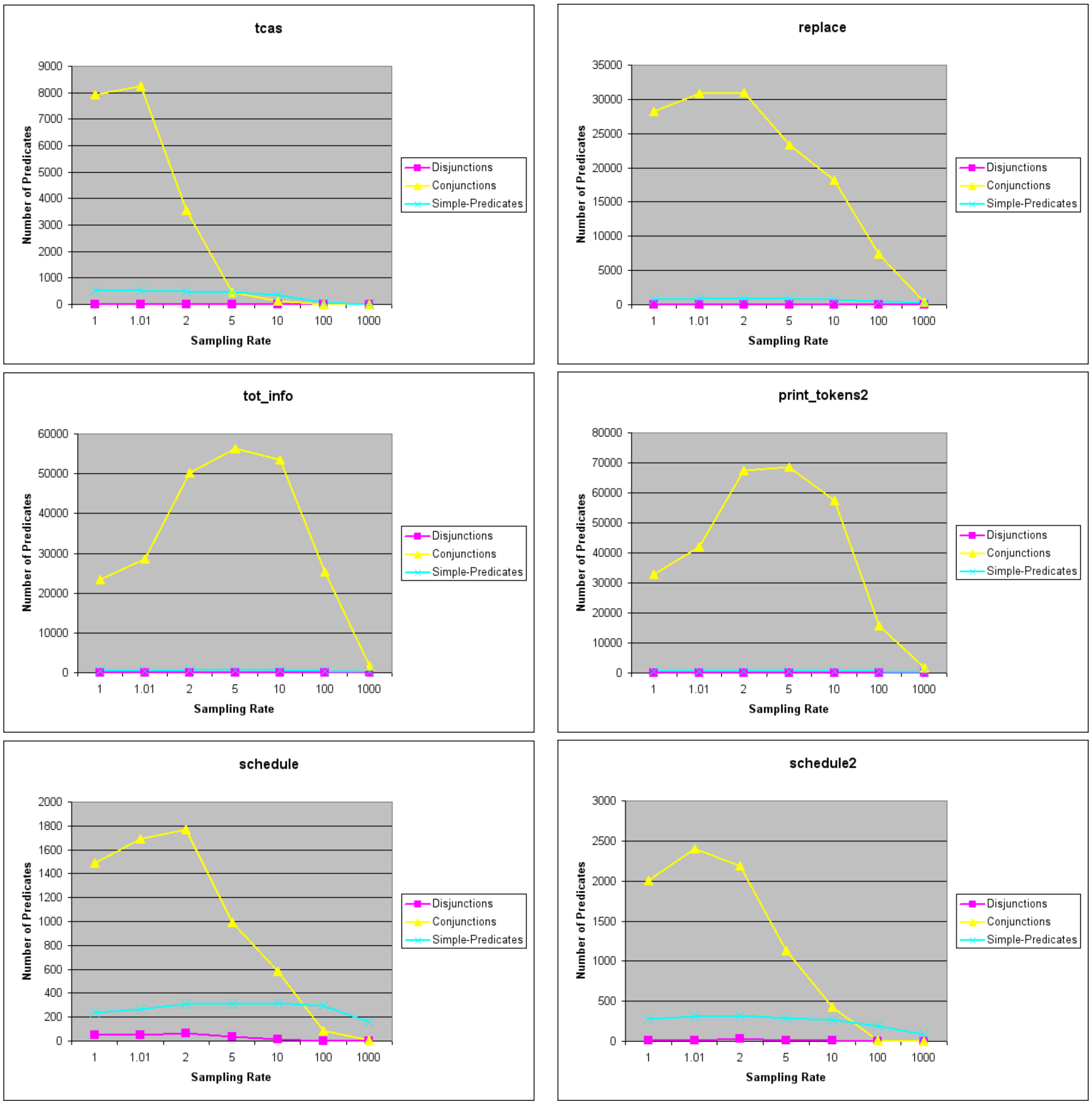

Figure 5. Sampling Rate vs. Number of Interesting Predicates

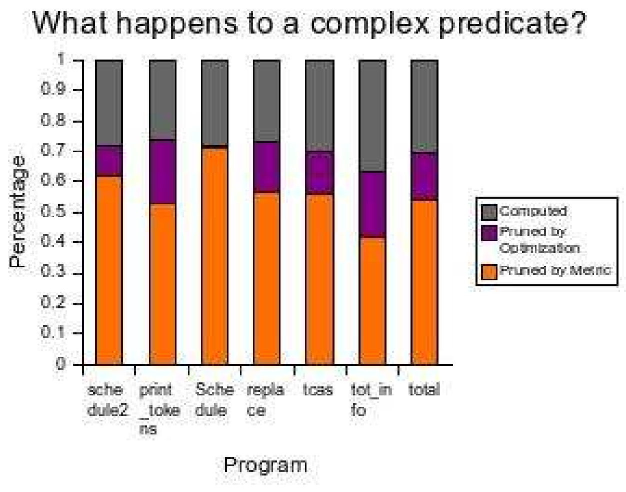

Figure 3. Improvement from pruning

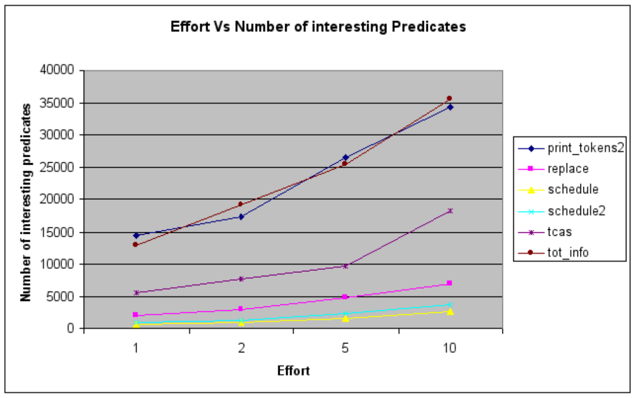

Figure 4. Variation of number of interesting predicates with $effort$

6.2 Improvement from Pruning

Figure 3 shows the percentage of complex predicates that are pruned by the optimizations discussed in subsection 3.3 and the metrics in section 4. For each program, the y-axis shows the contribution of the two kinds of pruning. On average, the usefulness metrics prune 54% of complex predicates and the optimizations in subsection 3.3 prune 15% of complex predicates. Only 31% of the complex predicates are actually computed.

6.3 Effect of the $Effort$ Parameter

Figure 4 has one curve for each program showing how the number of interesting predicates (Definition 3.3) varies at four different values - 1, 2, 5, 10 for $effort$. As expected, as $effort$ increases more predicates are evaluated and so more interesting predicates are found. This experiment serves as a sanity check for the implementation.

6.4 Effect of Sampling Rate

The dependence between sampling rate and the number of interesting predicates (both complex and simple) is plotted in Figure 5. Figure 5 has one chart per program with sampling rates in the $x-$axis and the average number of interesting conjunctions, disjunctions and simple predicates in the $y-$axis. The number of interesting disjunctions is always very low (order of tens) compared to interesting conjunctions. So the plot for interesting complex predicates closely follows the plot for conjunctions. At sampling rates higher than 10, there is a sharp drop in the number of interesting conjunctions. This is because with a sampling rate of N, the chance of observing a complex predicate is close to $\frac{1}{N^2}$ [5]. Despite the sharp drop, the number of interesting conjunctions is still comparable to the number of interesting simple predicates. This shows that sparse random sampling is not a significant detriment in finding interesting complex predicates.

A puzzling trend in Figure 5 is that the number of interesting conjunctions increases for a brief interval before dropping off. This trend is consistent across all programs. This can be due to two reasons:

1. Consider the *Increase* score (Eqn 1). Sampling may be reducing the number of *observed* runs in which a conjunction was true without affecting the *true* runs. This could happen because we do not collect sampled data directly but use scripts to downsample a data set collected with no sampling. Because of binarization of counts, downsampling of a count from 100 to 99 does not affect the score whereas downsampling from 1 to 0 affects the score.
2. The script that does downsampling uses the standard pseudo random number generator. Usually for experiments that use such random data, the values are averaged over multiple trials to get a confident estimate of the results. We weren't able to conduct multiple trials because of time constraints.

7. Related Work

Daikon [3] detects invariants in a program by observing values computed by it. It can compute complex invariants by combining program variables and operators like sum, max etc on collection (e.g. array) objects. Invariants are predicates that must be true in correct executions. [2] extends the work to compute implications of the form a $\Longrightarrow$ b. Our project is different from this in two ways. First, the data we have (bit vector of predicate counts) is different from what Daikon uses (values of variables at different points). So the fundamental techniques in our approach and [2] are different. Secondly, our project also aims at processing a sparse random sample of predicate values, whereas [3] requires complete execution traces. Diduce [4] is inspired by Daikon and identifies predicates that are true in failed runs. [9] is a statistical debugging tool similar to CBI. Neither of these approaches try to construct complex predicates.

8. Conclusion

We have demonstrated that complex predicates are useful predictors of bugs. Our experiments show qualitative and quantitative evidence that complex predicates can improve the current statistical analysis used by CBI. We describe two optimizations that make the task of computing complex predicates feasible. First is a numeric estimate on the upper bound of the score of a complex predicate. The second is a metric that quantifies the usefulness of a complex predicate. Even after these, the computational complexity is still high and requires further optimizations. The metrics described in section 4 help reduce the number of spurious predicates. But the metrics are not perfect as good bug predictors are still swamped by less useful ones. The algorithm described in [11] may solve this problem as it was designed with the goal of handling multiple predictors for the same bug.

[5] it is not exactly $\frac{1}{N^2}$ because of short circuiting boolean operations

References

[1] H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 342–351, 2005.

[2] N. Dodoo, A. Donovan, L. Lin, and M. D. Ernst. Selecting predicates for implications in program analysis. 2002.

[3] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2006.

[4] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, New York, NY, USA, 2002. ACM Press.

[5] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 191–200, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[6] A. Lal, J. Lim, M. Polishchuk, and B. Liblit. Path optimization in programs and its application to debugging. In P. Sestoft, editor, *15th European Symposium on Programming*, Vienna, Austria, Mar. 2006. Springer.

[7] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, Chicago, Illinois, June 12–15 2005.

[8] B. R. Liblit. *Cooperative Bug Isolation*. PhD thesis, University of California, Berkeley, Dec. 2004.

[9] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. Sober: statistical model-based bug localization. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 286–295, New York, NY, USA, 2005. ACM Press.

[10] E. W. Weisstein. Boolean function http://mathworld.wolfram.com/BooleanFunction.html. Dec.20 2006.

[11] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken. Statistical debugging: Simultaneous identification of multiple bugs. In W. Cohen and A. Moore, editors, *Proceedings of the 23rd International Conference on Machine Learning*, Pittsburgh, Pennsylvania, June 26–29 2006.