Protocol Inference Using Static Path Profiles

Murali Krishna Ramanathan[1], Koushik Sen[2],
Ananth Grama[1], and Suresh Jagannathan[1]

[1] Department of Computer Science, Purdue University
{rmk,ayg,suresh}@cs.purdue.edu
[2] Electrical Engineering and Computer Science,
University of California, Berkeley
ksen@eecs.berkeley.edu

Abstract. Specification inference tools typically mine commonalities among states at relevant program points. For example, to infer the invariants that must hold at all calls to a procedure p requires examining the state abstractions found at all call-sites to p. Unfortunately, existing approaches to building these abstractions require being able to explore all paths (either static or dynamic) to all of p's call-sites to derive specifications with any measure of confidence. Because programs that have complex control-flow structure may induce a large number of paths, naive path exploration is impractical.

In this paper, we propose a new specification inference technique that allows us to efficiently explore statically all paths to a program point. Our approach builds *static path profiles*, profile information constructed by a static analysis that accumulates predicates valid along different paths to a program point. To make our technique tractable, we employ a summarization scheme to merge predicates at join points based on the frequency with which they occur on different paths. For example, predicates present on a *majority* of static paths to *all* call-sites of any procedure p forms the pre-condition of p.

We have implemented a tool, MARGA, based on static path profiling. Qualitative analysis of the specifications inferred by MARGA indicates that it is more accurate than existing static mining techniques, can be used to derive useful specification even for APIs that occur infrequently (statically) in the program, and is robust against imprecision that may arise from examination of infeasible or infrequently occurring dynamic paths. A comparison of the specifications generated using MARGA with a dynamic specification inference engine based on CUTE, an automatic unit test generation tool, indicates that MARGA generates comparably precise specifications with smaller cost.

1 Introduction

The importance of clearly defined specifications to software development, maintenance, and testing is well-understood. Model-checking [5,14,26], type systems [7,9,10], typestate interpretation [17,13] and related static analyses [26]

M. Alpuente and G. Vidal (Eds.): SAS 2008, LNCS 5079, pp. 78–92, 2008.

are valuable only when proper specifications are available. The absence of specifications can also lead to improper reuse of program components and weaken the effectiveness of testing mechanisms[11,12,22].

In some cases, specifications are relatively easy to express (e.g., procedure p must always be called after data structure d is initialized), or are well-documented (e.g., a call to `socket` must be present before a call to `bind` and the return value of `socket` must be checked for erroneous conditions). In many cases, however, specifications are unknown, and even when available, are often informal.

To remedy this issue, there has been much recent interest in devising techniques that automatically extract program properties by mining program behavior. The effectiveness of mining critically depends on a sufficient number of use cases that can be examined. For example, if we are interested in inferring the pre-conditions that must hold prior to a call to a procedure p, we benefit by examining multiple calls to p. The more calls analyzed, the greater the likelihood we can effectively distinguish between predicates present at these calls that are truly part of p's specification from those that, although present, are nonetheless irrelevant. In general, the confidence in a mined pre-condition is significantly higher if it is observed in 90,000 out of 100,000 occurrences, compared to when it is observed in 9 out of 10 occurrences, even though the underlying percentage of occurrences are the same [15].

With a sufficient number of test cases, dynamic mining techniques can generate execution traces which contain a potentially large number of calls to p. Unfortunately, the cost to generate these traces may be high if we wish to ensure that these traces define a comprehensive enumeration of all possible executions of the program [11,12,22]. On the other hand, static techniques can only rely on static properties to identify call points; for example, if a call to p occurs at m different static program points in the source, only the predicates present at those m points can be mined. On the benchmarks used in our study, we observed that on an average, 80-85% of the procedures in these benchmarks are not invoked more than five times statically.

Furthermore, existing static techniques do not take into account the number of paths leading to different call-sites of the same procedure. Consider a predicate π that occurs at one call-site c_1, but which is absent at another call-site c_2 of the same procedure p. Static inference techniques would naturally deduce that π is not part of p's pre-condition. However, the number of paths leading to c_1 may significantly be greater than c_2 (e.g., c_2 may be part of an infrequently occurring error-inducing path). Indeed, it may be precisely the absence of π at c_2 that leads to an error.

The underlying premise of our work is that we can effectively apply the benefits of a dynamic analysis (i.e., generating a desired quantity of data for the purposes of mining) to a static specification mining algorithm. However, exploring all paths and generating the traces associated with each path statically has two significant disadvantages: (a) there are an exponential number of paths that would need to be examined, and (b) if only a subset of all paths are explored, then this approach has the same disadvantage of incompleteness common to any dynamic mining strategy.

Our main contribution in this paper is the development of an intelligent comprehensive path enumeration and summarization scheme that does not lead to exponential time and space costs. This goal is achievable because we are interested in deriving properties that are not path specific, but merely valid over a majority of the paths examined.

We define an inter-procedural, path-based static analysis that collects a set of program predicates that define potential pre-conditions to procedure calls. If procedure p has pre-condition π, it means that that all the predicates comprising π should hold before any call to p. These predicates can encode control flow (e.g., a call to `bind` must be preceded by a call to `socket`) as well as dataflow properties (e.g., the return value of `socket` is always validated before a call to `bind`). To compute pre-conditions, we analyze the predicates present along each control flow edge in the program's control-flow graph. At any join point j, where multiple paths merge, we keep track of the *number* of paths, n_j, leading to the join point and the number of times a predicate π is valid on these n_j paths. This information, which we refer to as a *static path profile*, is transferred to outgoing edges, and the process repeated until all control flow edges are traced. We use procedure summaries to make the approach scalable for inter-procedural analysis. To compute the pre-conditions of a procedure p, we take the cumulative sets of predicates associated with its different call-sites and their associated path profiles to derive the required specifications.

We have implemented a tool, MARGA, for computing path profiles for C programs. Note that an essential assumption underlying our approach is that the probability of occurrence of a dynamic path is likely to be equal to that of a static path. Clearly, such an assumption need not hold in general. Static paths may be infeasible, i.e., not be traversable under any dynamic execution. Similarly, a path that occurs frequently statically may occur infrequently dynamically because there may be stringent runtime conditions that dictate when the path can be traversed that are not captured by a static analysis. Conversely, a path that occurs frequently dynamically (e.g. the back edge of a long-lived loop) may occur infrequently statically. Fortunately, for the purposes of specification inference, we demonstrate that static path profiling is robust against inaccuracies introduced by failing to recognize (statically) infeasible, or infrequently occurring static and dynamic paths.

To support this claim, we have also implemented a dynamic specification inference engine that mines comprehensive dynamic executions of the program generated by CUTE [22], an automatic test generation tool. A comparison of the specifications inferred by MARGA and the dynamic inference engine reveals that infeasibility of program paths (or lack of correlation between probabilities of static *vs.* dynamic paths) has little impact on the quality of the specifications generated.

Bugs in programs present another challenge to specification inference since they may invalidate correct predicates from a specification or introduce incorrect ones. Test generation tools can help identify commonly occurring bugs since such bugs by definition must occur on many dynamic paths. Because these bugs

can be fixed, we assume programs are *mostly* free of errors. Bugs that are found on infrequently occurring dynamic paths are not always captured by unit testing. However, the paths on which these bugs occur must therefore necessarily correspond to profiled static paths with small weights. Consequently, the influence of these bugs on derived specifications is small.

Our experimental results using MARGA show that the analysis (a) can effectively infer specifications even for procedures with a small number of statically apparent call-sites; (b) exhibits fewer false negatives compared to static specification inference techniques that do not take path profiles into consideration, and (c) displays precision closer to that of an exhaustive dynamic path exploration technique.

2 Motivation

Dynamic specification inference techniques suffer from the problem of *under approximation* i.e., a set of predicates $\Theta_u = \{\pi_1, \pi_2, \ldots \pi_n\}$ is declared to hold before a call to procedure p even when only a subset of the predicates found in Θ_u are valid elements of p's pre-condition set; this is possible because not all possible paths to the call may have been examined, and these unexamined paths may invalidate the inclusion of some of the π_i in Θ_u.

Similarly, static specification inference techniques suffer from the problem of *over approximation*, i.e., a set of predicates Θ_o is considered to hold before a call to procedure p, even though other predicates (not present in Θ_o) should also be included; this is possible because a particular predicate that cannot be proven to hold along a certain path may result in its omission from Θ_o, even if that path is infeasible (e.g., the path follows a branch that could never be taken) or erroneous.

We elaborate on these points using the example shown in Figure 1. Before every call to procedure p, there are certain predicates that hold. For example, in Figure 1(a), there are two call-sites to p. There are two paths, labeled `1` and `2`, to one of the call-sites; on path `1`, predicate π_1 holds and on path `2`, predicates π_1 and π_2 hold. There are three paths leading to the other call-site to p (the call-site on the right of Figure 1(a)); these paths are labeled `3, 4` and `5`, with predicates π_1 and π_2 valid on paths `3` and `4`, and π_2 valid on path `5`. In a dynamic analysis scheme, if the paths `2, 3` and `4` are the only ones traversed, we may erroneously conclude that both π_1 and π_2 hold always before a call to procedure p. Note that this case is difficult to distinguish from the scenario

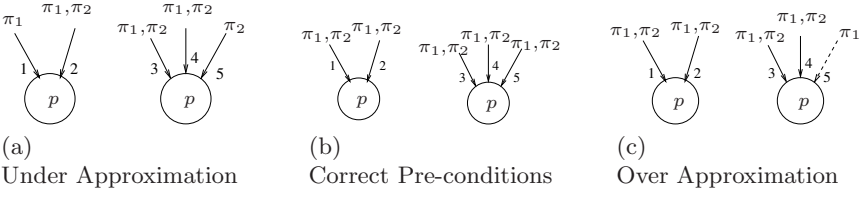


Fig. 1. An example illustrating under- and over-approximation of predicates

```
399 add_listen_addr(ServerOptions *options, char *addr, u_short port)
      ...
403   if (options->num_ports == 0)
404     options->ports[options->num_ports++] = SSH_DEFAULT_PORT;
      ...
407   if (port == 0)
408    for (i = 0; i < options->num_ports; i++)
409     add_one_listen_addr(options, addr, options->ports[i]);
410   else
411     add_one_listen_addr(options, addr, port);
```

Fig. 2. Motivating Example for over-approximation from `openssh-4.2p1`

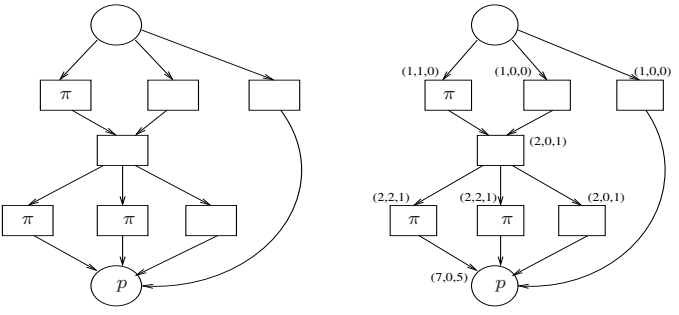
illustrated in Figure 1(b) where π_1 and π_2 *indeed* form the precondition for p. Ensuring that the paths `1` and `5` in Figure 1(a) are traversed depends upon the comprehensiveness of the test suite.

The problem of *over approximation* is illustrated in Figure 1(c). Here, there is one infeasible path (path `5`) to a call-site of p. A typical static analysis would conclude that one call to p (through paths `1` and `2`) has a set of predicates that include π_1 and π_2. Because of the absence of π_2 on the infeasible path `5`, the analysis would conclude that the other call to p (accessed through paths `3, 4` and `5`) does not include π_1 and π_2. Thus, given only two (static) calls to p, no statistically meaningful determination of p's pre-conditions can be made.

We provide further motivation for over approximation using a real-world example – statically deriving a specification for the `bind` system call in the Linux `socket` library. Part of the documented specification is that the address (second parameter to `bind`) corresponds to a specific address family (e.g., AF_UNIX, AF_INET). There are eight call-sites of `bind` in `openssh-4.2p1` of which *all* paths to five of the call-sites satisfy this specification. However, as shown in Figure 2, there exists a path to one of the call-sites where the address family is not set (`add_one_listen_addr` is not invoked). This happens when both `port` and `options->num_ports` are 0. This path is infeasible since both `port` and `options->num_ports` cannot be 0 simultaneously (line 404). Nevertheless, without the assistance of a theorem prover, static mining implementations must conservatively conclude that this is indeed a feasible path, and thus would be unable to conclude that the address family must be set prior to the call to `bind`. Using static path profiles, on the other hand, we will correctly weight the likelihood of this path occuring, and will not take into serious consideration the absence of the assignment to the address family.

3 Deriving Specifications

A simple technique to derive specifications is to trace each path in the program and then infer the set of valid pre-conditions from the traced paths. Consider the example shown in Figure 3(a). There are seven paths on total to a call-site



(a) Full Path Exploration (b) Static Path Profiles

Fig. 3. Illustrative example. Rectangles indicate predicates, circles indicate call-sites. Empty rectangles/circles indicate arbitrary predicates/procedure calls.

of some procedure p. If every path is traced statically, it is clear that among five out of the seven paths, the predicate π holds and is a precondition for p with confidence 71.42%. Although this scheme is simple, the cost associated with tracing each path is exponential in the number of edges in the program.

The key insight to our approach is that obtaining aggregate information associated with multiple paths is sufficient for generating interesting pre-conditions. Knowing the specific paths in which π holds is uninteresting from the perspective of specification inference. A static path profile is the cumulative information of predicates that hold across all possible paths to a specific call-site.

Path information is collected by examining the program's control-flow graph. Each node v in the CFG is annotated as a three tuple (n_v, m_v, q_v) for every predicate π under consideration, where the definition of the tuple components is as follows:

- n_v is the total number of paths leading to v,
- $m_v = \begin{cases} n_v, \text{if predicate } \pi \text{ holds at } v \\ 0 \text{ otherwise} \end{cases}$
- $q_v = \Sigma_u \, \texttt{max}\,(m_u, q_u)$ where $u \in \text{predecessor}(v)$ in the CFG.

At any given node, we can derive the number of paths any predicate π holds by observing the three-tuple (n_v, m_v, q_v) associated with the predicate π at that node. The number of predicates examined at a node is directly proportional to the number of variables. Intuitively, q_v specifies the number of paths through v in which the predicate is valid. If a predicate π is valid on some number of incoming paths *upto* node v, but in addition also happens to be asserted *at* v, it is clear π holds on all paths *through* v ($m_v = n_v$). The nodes downstream from v decide the number of paths on which π holds using q_v and m_v. If the number of paths for which π holds is i, then $i = \texttt{max}\,(m_v, q_v)$; the fact that predicate π holds on i paths is denoted as π_i.

For example, consider the annotated graph counterpart of Figure 3(a) in Figure 3(b) associated with predicate π. Let one of the two nodes annotated

`(2,0,1)` be v. This annotation denotes the fact that there are two possible paths to node v, predicate π is not explicitly valid at v, and the total number of paths on which π is valid is one (written π_1).

Loops pose complications for building path profiles because they represent a potentially infinite set of executions. To make our approach tractable, we perform a simple fixpoint calculation to compute the path profile for back-edges in loops. Initially, we assume the back-edge does not contribute to the profile weights of any path found within the loop. In subsequent iterations of the analysis, the back edge on the loop contributes exactly once to the profile weights, albeit with the predicates being derived propagated through the back edge multiple times. Since the computation of the tuple is monotonic (since q_v computes the maximum of the profiles of its predecessors which is bounded by the number of paths in the loop body that include the back-edge), the analysis is guaranteed to converge.

The annotation marking mechanism must also take into account nodes in the control-flow graph that represent call-sites (e.g., the node labeled p). Path profiles distinguish between incoming and outgoing annotations. The incoming annotation in p's case is `(7,0,5)`. Incoming annotations are used to generate preconditions for p. Thus, to infer the pre-condition for p requires no inspection of p's body. In this case, π_5 holds true at node p, i.e., five paths of the total set of paths have π to be true. Outgoing annotations (not shown in the figure), on the other hand, capture path profile summary information. The summary information for some procedure p gives the number of paths within p for which the predicate holds upon exit from p, which in turn is given by the annotation at p's *return* node. Summary information is used to define incoming annotations for other call-sites downstream in the graph. We elaborate on this point in the next section.

4 MARGA : Implementation Details

We have implemented a tool name MARGA based on the above approach. It takes as input the program source and a user-defined confidence threshold for determining when a predicate should form part of a pre-condition, and produces as output pre-conditions (i.e., a set of predicates) for every procedure. These pre-conditions indicate the conditions that must hold prior to any call of the associated procedure.

We first generate the control-flow graph for each procedure using Codesurfer [3]. The resulting graph is processed using the algorithm given in Figure 4. The number of paths to each node in the graph is first computed. Subsequently, the q value for the node is computed for each predicate by considering all its parent nodes. If the node is a call-site, then the *procedure summary* associated with that call is also added to the set of predicates that will flow into other adjacent nodes in the graph. The procedure summary is the summary of predicate information along with the total number of paths and the number of paths for which each predicate holds at the *return* node of the procedure. This process is repeated until a convergent path profile for the loop back-edge is

```
procedure BUILDPREDICATES
  ▷ Input: G(V,E) , directed, acyclic CFG of α ; V is topologically sorted;
  ▷ Output: Annotated Graph G
  ▷ Auxiliary Information:
       predicates(u): predicates generated at u; flow(u): set of predicates valid at u;
       precond(u): set of predicates that are used for generating preconditions associated with procedure at u;
       CALLSITE(u): true if u is the return node from procedure α;

1  iterate ← true
2  while iterate do
3    iterate ← false
4    for each node u = 1 to |V|
5      oldflow ← flow(u)
6      for all predecessors v of u
7          n_u ← n_u + n_v
8          flow(u) ← flow(u) ∪ predicates(v)
9          for each predicate π in flow(v)
10             q_u(π) ← q_u(π) + max(m_v(π), q_v(π))
11             m_u(π) ← 0
12     flow(u) ← flow(u) ∪ predicates(u)
13     for each predicate π in data_predicate(u)
14         m_u(π) ← n_u
15     if CALLSITE(u) is true then
16         precond(u) ← flow(u)
17         flow(u) ← flow(u) ∪ proc_summary[func(u)]
18     if RETURN(u) is true then
19         proc_summary[α] ← flow(u)
20     if oldflow ≠ flow(u) then iterate ← true
```

Fig. 4. Algorithm for building predicates

```
procedure GETPRECONDITIONS
   ▷ Input: α: a procedure in the program;
                C = {c_1, c_2, ...c_n} is the set of call sites of α;
                β = user-defined threshold for generating preconditions
   ▷ Output:set of preconditions for α
1  for each node c_i
2      for each predicate π in precond(c_i)
3          q_t(π) ← q_t(π) + q_{c_i}(π)
3          n_t(π) ← n_t(π) + n_{c_i}
4      flow_t ← flow_t ∪ precond(c_i)
5  for each predicate π in flow_t
6      if q_t/n_t > β
7          preconditions[α] ← preconditions[α] ∪ {π}
```

Fig. 5. Generate preconditions

computed.Yet another fix point iteration is performed to ensure that dependencies crossing procedure boundaries (as given by the *procedure summary* and *return*) are completely captured.

At the end of the fixed point calculation, the algorithm shown in Figure 5 is executed to obtain the pre-conditions associated with each procedure in the program. To generate the pre-condition for a procedure p, each call-site of p is considered. The predicates that can be used in computing the preconditions from each of these call-sites is extracted and the total number of paths in which the predicate holds (q_t) across all call-sites is computed. Similarly, the total number of paths to all call-sites (n_t) is also calculated. If the ratio of the number of

Table 1. Benchmark Information

Source	Version	LOC	CFG nodes	Procedure count	Avg. paths to call-sites	Total Specifications	Analysis time (s)
`zebra`	0.95a	183K	145K	3342	4721.64	1323	555
`apache`	2.2.3	273K	102K	2079	7281.30	676	561
`openssh`	4.2p1	68K	88K	1281	7175.75	619	501
`osip2`	3.0.1	24K	34K	666	4028.32	158	104
`procmail`	3.22	9K	16K	298	33696.15	120	297
`buddy`	2.4	10K	10K	173	5653.23	133	140

paths on which a predicate holds compared to the total number of paths at all call-sites is greater than β, a user-defined threshold, the predicate is added to the pre-condition for p.

5 Experiments

We validate the idea of using static path profiles on selected benchmark sources to demonstrate scalability and effectiveness. We extract pre-conditions for six sources: `apache`,`buddy`, `zebra`, `openssh`, `osip2`, and `procmail`. Specific details about these benchmarks are provided in Table 1. The size of the benchmarks varies from 9K to 273KLOC. Since default configurations are used to compile these sources, we believe that the number of control flow nodes represents a more reliable indicator of effective source size than lines of code. The number of control flow nodes ranges from 10K to 143K. We also present the number of user-defined procedures examined in the table.

We have implemented our tool in C++ and have performed our experiments on a Linux 2.6.11.10 (Gentoo release 3.3.4-r1) system running on an Intel(R) Pentium(R) 4 CPU machine operating at 3.00GHz, with 1GB memory. The time taken for performing the analysis is presented in Table 1.

5.1 Quantitative Assessment

We derive pre-conditions containing three different types of predicates – assignment, comparison and precedence. As their names suggest, assignment predicates reflect the assignment of values (or results of procedure calls) to variables; comparison predicates include six kinds of logical comparison operations ($>$, $<$, $\neq$, $=$, $\geq$ and $\leq$) between variables and/or constants; a precedence predicate is an ordered sequence of procedures whose calls must precede the call to the procedure being examined. The total number of pre-conditions generated for procedures, where the predicates are valid on at least 70% of the paths is given in Table 1. The size distribution (number of predicates within a pre-condition) for the generated pre-conditions is given in Figure 6. Among the generated pre-conditions, the size of the predicate set is less than two for a majority of the procedures. For example, observe that approximately 97% of the procedures in `apache` have

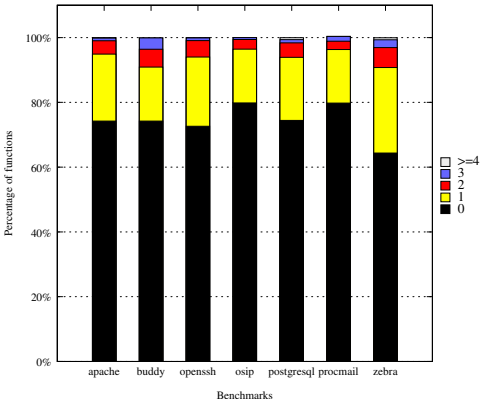


Fig. 6. Predicate distributions

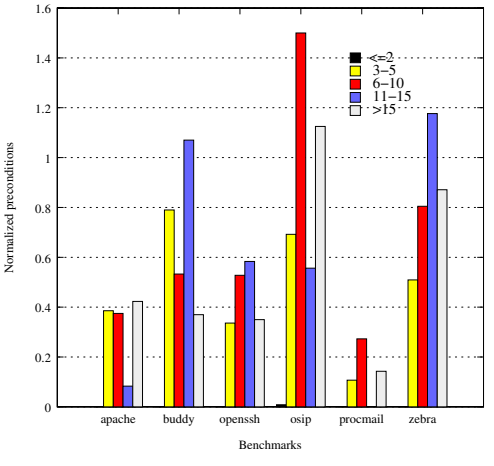


Fig. 7. Comparison with non-profile based inference

fewer than two predicates in their pre-conditions. The predicate size distribution display a similar pattern for different types of predicates.

We experimentally compare our approach with a non-profile based inference mechanism that does not leverage path profiles [21]. Briefly, the comparison metric is an analysis that requires a predicate to be satisfied along all paths to a call-site in order to be a valid candidate for inclusion as part of a procedure call's pre-condition. After accumulating the predicates at each call-site, we declare a predicate as a pre-condition, if the predicate is valid in at least the user-defined threshold percentage of call-sites. We use the same user-defined threshold (70%) in deriving the predicates, i.e., if a predicate is valid in seven out of 10 call-sites, we declare the predicate as a pre-condition in the non-profile based inference scheme. In the path-profiling approach, we declare a predicate to be a pre-condition if it is valid in 70% of the *paths* to call-sites of the procedure.

Figure 7 presents the percentage of pre-conditions derived by non-profile based specification inference as compared to those derived using static path profiling. For example, for procedures with three to five call-sites in `zebra`, the former discovers roughly only half the predicates discovered by the path profile analysis. As expected, for procedures with fewer than three call-sites, the non-profile based inference scheme is not able to derive any pre-conditions. For example, in the case of `openssh`, no pre-condition is derived for procedures that have fewer than three statically apparent call-sites.

We also observe that in many cases, the set of pre-conditions generated with the non-profile based inference is a proper *subset* of the pre-conditions generated using the static path profiling approach. This is consistent with our expectation that typical static analyzes can lead to over approximation by eliminating valid predicates from pre-conditions. In some cases, however, such as `osip` or `zebra`, this hypothesis does not hold. Path profiling weights predicates based on the number of paths on which they hold across all call-sites. Consider a predicate π which occurs on k paths at n call-sites to procedure p. Suppose that paths are not evenly distributed among these n call-sites. If on m call-sites, π occurs on all paths, and m is greater than the threshold cutoff, the non-profile based inference will record π as a valid pre-condition. However, if the number of paths that flow into these m call-sites is much less than k, then the path profile analysis will nonetheless not include π as part of p's pre-conditions. In other words, a predicate that does not hold on a majority of paths may still hold on the paths to a majority of call-sites. Path profiling thus provides finer control over both the inclusion and exclusion of predicates than non-profile based inference.

5.2 Qualitative Assessment

We want to identify the impact of infeasible paths and approximations introduced by static path weights in the program on the quality of the specifications inferred. To do so, we compare our approach with a dynamic inference mechanism. Rather than using an existing test-suite to generate dynamic traces, we use CUTE, an automatic test generation tool, that provides extended coverage of the program, and thus helps reduce the possibility of under-approximation (compared to other dynamic analysis systems) as described in Section 2.

The test generation process initially runs with some random input and collects constraints $\{C_1, ..., C_{k-1}, C_k\}$ symbolically along the execution. To explore a previously unexplored path, a new input is generated that satisfies the constraints $\{C_1, ...C_{k-1}, \neg C_k\}$. If this path was explored earlier, then the set of constraints $\{C_1, ...\neg C_{k-1}\}$ is used to explore a different path. This process repeats until all paths in the program are explored. There are several issues that must be handled by the input generation process. Most importantly, when it becomes difficult to reason with symbolic constraints, concrete values from the program execution replace symbols to ensure progress of the test input generation process. We refer the reader to [22] for a more detailed description.

We track the predicates along different execution paths of the program and for each call to a procedure, group the set of predicates that precede it from the

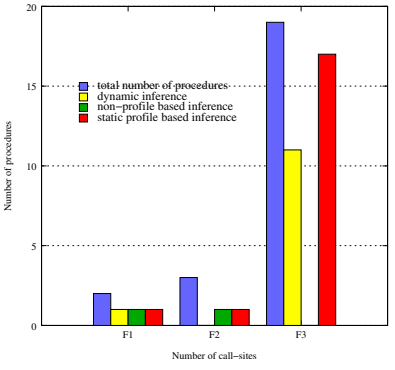


Fig. 8. Correctness of different inference schemes

start of the execution. Thus, across multiple executions, we would generate many such groups of predicates. To generate the precondition for the procedure, we apply frequent item-set mining [6]. The frequently occurring predicates across all the groups form the precondition.

We performed our comparison on `buddy`, an open source package that implements operations over Binary Decision Diagrams (BDD). We ran CUTE for a bounded number of iterations (1000), which took approximately 30 minutes, and in that process collected specifications for 24 procedures. Of these 24 procedures, only two procedures(F1) had more than 10 call-sites, three procedures (F2) had call-sites between five and 10 and the remaining 19 procedures(F3) had less than five call-sites. Using existing documentation, and manual inspection, we computed a reference specification for each of these procedures.

Figure 8 presents the results associated with our qualitative analysis. We applied three different schemes, (a) dynamic inference (b) non-profile based inference, and (c) path-profile based inference on this benchmark. For the set of two procedures in F1, all techniques provide similar precision and were able to detect preconditions correctly for one procedure. Under-approximation confounds the precision of dynamic inference for the set of three procedures in F2. The analysis for the procedures in F3 is more interesting. Because of the lack of frequency of call-sites for the procedures in this set, non-profile based static inference is ineffective in producing specifications with any degree of confidence. In contrast, path-profile based inference correctly identified the correct specification for 17 of the 19 procedures. Surprisingly, under-approximation still poses a problem even for a comprehensive test generation tool like CUTE; it failed to correctly generate specifications for 7 of the procedures that were successfully analyzed using static path profiling.

6 Related Work

Many interesting static mining approaches exist for specification inference. Kremenek *et al.* [16] develop a inference mechanism based on using factor graphs.

Even though, many useful specifications were generated, the approach requires either machine learning or user specifications to generate initial annotation probabilities, employs naming conventions for improving accuracy and is domain-specific. Ramanathan *et al.* [21] present an annotation-free approach to infer data flow specifications using frequent item-set mining and control flow specifications (precedence relations [20]) using sequence mining. This approach is path-sensitive, but does not take static path profiles into account: if a predicate does not hold at a majority of call-sites to a procedure, it is not included as part of the procedure's pre-condition. Shoham *e al.* [23] propose an approach for client-side mining of temporal API specifications based on static analysis. Li and Zhou present PR-Miner [18], a tool that relies on mining to identify frequently occurring program patterns. As this approach is not path-sensitive, spurious specifications can be generated even if a predicate holds in *at least* one path leading to a majority of call sites. Mandelin *et al.* [19] present a technique for synthesizing jungloid code fragments automatically based on the input and output types that describe the code and is useful for reusing existing code. An automatic specification mining technique that uses information about exceptions to identify temporal safety rules is presented in [25]. Because none of the above techniques performs mining on generated paths, the confidence in the derived specifications is statistically low. Due to the ability of our approach to simulate dynamic behavior and succinctly maintain data that covers all possible program paths, we are able to derive useful specifications with high confidence.

Ball and Larus [4] propose an approach for efficient path profiling. In their approach, dynamic runs of a program are profiled to gather information about different path executions. Recently, Vaswani *et al.* [24], present a scheme to identify a subset of interesting paths and use a compact numbering scheme using arithmetic coding techniques. Our approach is motivated by the algorithm presented by Ball and Larus [4] and is applied statically.

Ammons *et al.* [1] perform specification mining by summarizing frequent interaction patterns as state machines that capture temporal and data dependencies when interacting with API's or abstract data types. An approach to debug derived specifications using concept analysis is subsequently proposed by Ammons *et al.* in [2]. Daikon [8] is a tool for dynamically detecting invariants in a program. Yang *et al.* [27] present a technique for automatically inferring temporal properties by exploring event traces across versions of a program. All these approaches critically rely on test input providing comprehensive coverage. Our approach is independent of test inputs and covers all possible program paths.

Acknowledgements

This work is supported in part by the National Science Foundation under grant CNS-509387. We also like to thank the anonymous reviewers for their constructive feedback.

References

1. Ammons, G., Bodik, R., Larus, J.: Mining specifications. In: Proceedings of POPL 2002, pp. 4–16 (2002)
2. Ammons, G., Mandelin, D., Bodik, R., Larus, J.: Debugging temporal specifications with concept analysis. In: Proceedings of PLDI 2003, pp. 182–195 (2003)
3. Anderson, P., Reps, T., Teitelbaum, T.: Design and implementation of a fine-grained software inspection tool. IEEE Trans. on Software Engineering 29(8), 721–733 (2003)
4. Ball, T., Larus, J.: Efficient path profiling. In: MICRO-29 (December 1996)
5. Ball, T., Rajamani, S.K.: Automatically validating temporal safety properties of interfaces. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 103–122. Springer, Heidelberg (2001)
6. Burdick, D., Calimlim, M., Flannick, J., Gehrke, J., Yiu, T.: Mafia: A performance study of mining maximal frequent itemsets. In: FIMI 2003 (2003)
7. Chin, B., Markstrum, S., Millstein, T.: Semantic type qualifiers. In: Proceedings of PLDI 2005, pp. 85–95 (2005)
8. Ernst, M., Cockrell, J., Griswold, W., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. IEEE TSE 27(2), 1–25 (2001)
9. Foster, J., Terauchi, T., Aiken, A.: Flow-sensitive type qualifiers. In: Proceedings of PLDI 2002 (2002)
10. Furr, M., Foster, J.: Checking type safety of foreign function calls. In: Proceedings of PLDI 2005 (2005)
11. Godefroid, P.: Compositional dynamic test generation. In: POPL 2007, pp. 47–54 (2007)
12. Godefroid, P., Klarslund, N., Sen, K.: Dart: Directed automated random testing. In: Proceedings of PLDI 2005, Chicago, Il, pp. 213–223 (2005)
13. Henzinger, T., Jhala, R., Majumdar, R.: Permissive interfaces. SIGSOFT Softw. Eng. Notes 30(5), 31–40 (2005)
14. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley, Reading (2004)
15. Kapadia, A.S., Chan, W., Moye, L.A.: Mathematical Statistics With Applications. CRC, Boca Raton (2005)
16. Kremenek, T., Twohey, P., Back, G., Ng, A., Engler, D.: From uncertainty to belief: Inferring the specification within. In: Proceedings of OSDI 2006 (2006)
17. Lam, P., Kuncak, V., Rinard, M.: Generalized typestate checking for data structure consistency. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, Springer, Heidelberg (2005)
18. Li, Z., Zhou, Y.: Pr-miner: Automatically extracting implicit programming rules and detecting violations in large software code. In: Proceedings of ESEC-FSE 2005 (September 2005)
19. Mandelin, D., Xu, L., Bodik, R., Kimelman, D.: Jungloid mining: Helping to navigate the api jungle. In: Proceedings of PLDI 2005, pp. 48–61 (2005)
20. Ramanathan, M.K., Grama, A., Jagannathan, S.: Path-sensitive inference of function precedence protocols. In: Proceedings of ICSE 2007 (May 2007)
21. Ramanathan, M.K., Grama, A., Jagannathan, S.: Static specification inference using predicate mining. In: Proceedings of PLDI 2007, pp. 123–134 (2007)
22. Sen, K., Marinov, D., Agha, G.: Cute: A concolic unit testing engine for C. In: Proceedings of ESEC-FSE, pp. 263–272 (2005)

23. Shoham, S., Yahav, E., Fink, S., Pistoia, M.: Static specification mining using automata-based abstractions. In: ISSTA 2007: International Symposium on Software Testing and Analysis, pp. 174–184 (July 2007)
24. Vaswani, K., Nori, A.V., Chilimbi, T.M.: Preferential path profiling: compactly numbering interesting paths. In: Proceedings of POPL 2007, Nice, France (January 2007)
25. Weimer, W., Necula, G.: Mining temporal specifications for error detection. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 461–476. Springer, Heidelberg (2005)
26. Xie, Y., Aiken, A.: Scalable error detection using boolean satisfiability. In: Proceedings of POPL 2005 (2005)
27. Yang, J., Evans, D., Bhardwaj, D., Bhat, T., Das, M.: Perracotta: Mining temporal api rules from imperfect traces. In: Proceedings of ICSE 2006 (May 2006)