ESP: Path-Sensitive Program Verification in Polynomial Time

Manuvir Das
Microsoft Research
manuvir@microsoft.com

Sorin Lerner
University of Washington
lerns@cs.washington.edu

Mark Seigle
University of Washington
seigle@cs.washington.edu

ABSTRACT

In this paper, we present a new algorithm for partial program verification that runs in polynomial time and space. We are interested in checking that a program satisfies a given temporal safety property. Our insight is that by accurately modeling only those branches in a program for which the property-related behavior differs along the arms of the branch, we can design an algorithm that is accurate enough to verify the program with respect to the given property, without paying the potentially exponential cost of full path-sensitive analysis.

We have implemented this "property simulation" algorithm as part of a partial verification tool called ESP. We present the results of applying ESP to the problem of verifying the file I/O behavior of a version of the GNU C compiler (gcc, 140,000 LOC). We are able to prove that all of the 646 calls to `fprintf` in the source code of gcc are guaranteed to print to valid, open files. Our results show that property simulation scales to large programs and is accurate enough to verify meaningful properties.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging; D.3.4 [**Programming Languages**]: Compilers; F.3.1 [**Theory of Computation**]: Specifying and Verifying and Reasoning about Programs

General Terms

Algorithms, Security, Verification.

Keywords

Path-sensitive analysis, dataflow analysis, error detection.

1. INTRODUCTION

In recent years, program analysis techniques have been used to build tools for partial verification [25, 11, 3, 10]. The programmer provides a description of a temporal safety property written as a finite state machine. An example of such a property is given in Figure 1. An analysis tool then tracks the state of the property FSM through a program. If the error state is never reached, the program obeys the safety property.

Previous work on partial verification has focused on path-sensitive analysis methods [3, 14]. These methods are accurate enough for verification because they are able to reason about branch correlations, which is usually necessary to control the number of false error reports generated during verification. However, the cost of path-sensitivity has limited the applicability of these methods to large programs. In this paper, we present a new path-sensitive method for partial verification that scales to large programs.

Path-sensitive analysis can be expensive because accurately tracking every branch in the control-flow of a program in which the execution state (e.g., values of variables) differs along the two branch paths may result in an exponential or infinite search space. However, given a particular property to be checked, it is likely that most branches in the code are not relevant to the property, even though they affect the execution state of the program. The trick is to identify and accurately track only relevant branches.

In this paper, we present "property simulation", a new method for partial verification that is based on the heuristic that a branch is likely to be relevant only if the property FSM transitions to different states along the arms of the branch. This heuristic leads to a path-sensitive algorithm that is sensitive to the "property state": symbolically evaluate the program, generating symbolic states that include both the execution state and the state of the property FSM. At a merge point in the control flow, if two symbolic states have the same property state, produce a single symbolic state by merging their execution states as in dataflow analysis. Otherwise, process the symbolic states independently as in path-sensitive analysis. This method avoids exponential blowup and captures relevant branching behavior.

We make the following contributions:

- We present a framework for inter-procedural property simulation. Particular algorithms of varying precision and complexity can be obtained by fixing the domain of execution states used in the framework.
- We focus on one particular instantiation of the frame-

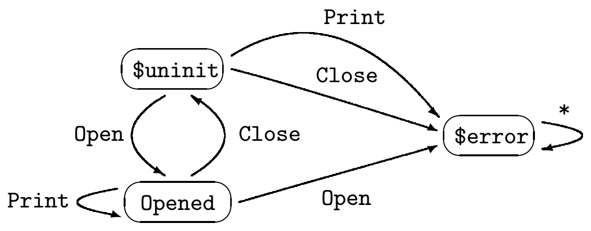

Figure 1: A temporal safety property that encodes correct usage of the stdio file output library.

work, in which the domain of execution states is chosen to be the constant propagation lattice. We show that in this case, inter-procedural property simulation terminates in polynomial time and space.

- We describe ESP, a system that uses a combination of scalable alias analysis and property simulation to verify that large code bases obey temporal safety properties. Property simulation can also be used to provide an accurate starting point for a system based on iterative refinement, such as SLAM [3].
- We present the results of a case study: verifying output file manipulations in the gcc compiler (taken from SPEC95) using ESP. Our results show that:
  - Property simulation is accurate. We are able to verify that all of the 646 calls to `fprintf` in gcc are guaranteed to print to valid, open files.
  - Property simulation is scalable. For each of the 15 files to which gcc writes its output, we are able to perform inter-procedural simulation of the source code of gcc (140,000 LOC) on average in 70secs and 50MB. To our knowledge, our analysis is the first method to have verified temporal safety properties of a program of this size.

The rest of this paper is organized as follows: In Section 2, we present a motivating example for our algorithm. In Section 3, we present intra-procedural and inter-procedural versions of property simulation. In Section 4, we describe the ESP verification system. In Section 5, we present our case study. We survey related work in Section 6, and conclude in Section 7.

2. EXAMPLE

In this section, we use an example to explain how certain types of programming errors can be detected using temporal safety properties. We use the example to motivate our property simulation algorithm.

Consider the simplified snippet of code from the gcc compiler shown in Figure 2. Suppose we are interested in determining whether this piece of code interacts correctly with the stdio library through calls to `fopen` and `fclose`. For instance, a file handle may be closed through a call to `fclose` only if it has previously been opened through a call to `fopen`, and it has not already been closed through a previous call to `fclose`. There is no mechanism in a language like C to express such usage rules in the type system, so that they may be enforced by a compiler (*cf.* [10]).

```
 void main(){
   if (dump)
     f = fopen(dumpFil,"w"); /* Open */
   if (p)
     x = 0;
   else
     x = 1;
l: if (dump)
     fclose(f); /* Close */
 }
```

Figure 2: A (simplified) snippet of code from a version of the gcc compiler.

One way to overcome this problem is to instrument the program by mapping calls to library functions to transitions in the property FSM from Figure 1. Program analysis techniques can then be applied to conservatively over-approximate the possible states into which the property FSM may be driven along all execution paths. If the error state is not encountered, verification of the program with respect to the property succeeds.

Example 1. Assume that calls to `fopen` and `fclose` are mapped to corresponding transitions `Open` and `Close` from the property FSM. The possible symbolic states inferred by three analyses at label `l` in Figure 2 are described below. Each symbolic state includes the possible states of the property FSM and the execution state of the program.

Path-sensitive analysis. There are four feasible paths to `l`, resulting in the states:

$$[\texttt{\$uninit}, \neg\texttt{dump}, \neg\texttt{p}, \texttt{x} = 1] \quad [\texttt{Opened}, \texttt{dump}, \neg\texttt{p}, \texttt{x} = 1]$$
$$[\texttt{\$uninit}, \neg\texttt{dump}, \texttt{p}, \texttt{x} = 0] \quad [\texttt{Opened}, \texttt{dump}, \texttt{p}, \texttt{x} = 0]$$

These states capture the correlation between the property state and `dump`, which is necessary to avoid following the true branch of the condition at `l` in the `$uninit` state.

Standard dataflow analysis. States are merged at join points in the CFG. Therefore information about `dump` is lost, and the single state produced at `l` is $[\{\texttt{\$uninit}, \texttt{Opened}\}]$. As a result the `Close` transition is processed from the `$uninit` state, leading to a false error report.

Property simulation. Along the true branch of the first conditional, the property state is changed from `$uninit` to `Opened`. Therefore, the two symbolic states from the first conditional are not merged. The second conditional does not affect the property state. Therefore, the four symbolic states arising from the second conditional are merged into two states at `l`: $[\texttt{\$uninit}, \neg\texttt{dump}]$ and $[\texttt{Opened}, \texttt{dump}]$. These states capture the correlation between the property state and `dump`, but drop the correlation between `p` and `x`, which is not relevant to checking the file output property. □

Property simulation is precise because it is designed to match the behavior of a careful programmer. Before calling a function that affects the property state, she must check the current state of the property FSM to ensure that the function call will not result in an error. However, the programming language provides no mechanism to express or check the property state. Instead, she uses conditionals to check the program state before the function call. In other words, the programmer maintains an implicit correlation between a given property state and the execution states under

```
global
  $Worklist : 2^N$
  $Info : E \to 2^S$

procedure Solve(CFG = $[N, E]$)
begin
  for each $e \in E$ do $Info(e) := \{\}$
  $Info(Out_T(n_{entry})) := \{[\$uninit, \top]\}$
  $Worklist := \{dst(out_T(n_{entry}))\}$

  while $Worklist \neq \emptyset$ do
    Remove a node $n$ from $Worklist$
    switch($n$)
      case $n \in Merge$:
        $ss = F_{mrg}(n, Info(In_0(n)), Info(In_1(n)))$
        Add($Out_T(n), ss$)
      case $n \in Branch$:
        $ss_T = F_{br}(n, Info(In_0(n)), true)$
        $ss_F = F_{br}(n, Info(In_0(n)), false)$
        Add($Out_T(n), ss_T$)
        Add($Out_F(n), ss_F$)
      case $n \in Other$:
        $ss = F_{oth}(n, Info(In_0(n)))$
        Add($Out_T(n), ss$)

  return $Info$
end

procedure Add($e, ss$)
begin
  if $Info(e) \neq ss$ then
    $Info(e) := ss$
    $Worklist := Worklist \cup \{dst(e)\}$
end
```

Figure 3: Intra-procedural property analysis.

which the property FSM is in that state. Property simulation makes this correlation explicit in the analysis.

In the example above, we assumed that there is a single file handle in the program, and that changes to its state can be identified syntactically. In practice, neither assumption is valid. In Section 4, we describe ESP, a system that uses a combination of scalable alias analysis and property simulation to track multiple stateful values.

3. PROPERTY SIMULATION

In this section, we describe property simulation in detail. We first present property analysis, a general framework for tracking property states and execution states using path-sensitive dataflow analysis. By varying a function α that groups together execution states, we obtain three dataflow analyses: a fully path-sensitive analysis, a standard dataflow analysis, and property simulation. Each analysis can be further instantiated to algorithms of particular precision and complexity by choosing a particular domain for execution states. We describe an instantiation of property simulation based on constant propagation. We show that for this case, property simulation runs in polynomial time and space.

3.1 Intra-procedural property analysis

In this subsection we describe a generic dataflow analysis, "property analysis", that computes the set of possible property states at all points in a single procedure program.

We assume a standard CFG with a distinguished entry node n_{entry}, merge nodes with exactly two predecessors,

$$F_{mrg}(n, ss_1, ss_2) = \alpha(ss_1 \cup ss_2)$$

$$F_{br}(n, ss, val) = \alpha(\{s' | s' = f_{br}(n, s, val) \wedge s \in ss \wedge es(s') \neq \bot\})$$

$$F_{oth}(n, ss) = \alpha(\{f_{oth}(n, s) | s \in ss\})$$

(a) Flow functions

$$\alpha_{cs}(ss) = ss$$

$$\alpha_{df}(ss) = \{[\textstyle\bigcup_{s \in ss} as(s), \bigsqcup_{s \in ss} es(s)]\}$$

$$\alpha_{as}(ss) = \{[\{d\}, \textstyle\bigsqcup_{s \in ss[d]} es(s)] | d \in D \wedge ss[d] \neq \emptyset\}$$

where $ss[d] = \{s | s \in ss \wedge d \in as(s)\}$

(b) Grouping function

Figure 4: Definitions of flow functions and the grouping function.

branch nodes with a single predecessor, a true successor, and a false successor, and computation nodes with a single predecessor and a single successor. We use appropriately named accessor functions to extract edge information from CFG nodes and vice versa (In, Out, src, dst).

We use the following domains: D is the (finite) set of states in the (deterministic) property state machine. D includes two distinguished states: `$uninit`, the initial state, and `$error`, the error state. S is the domain of *symbolic states*. A symbolic state is a pair containing an *abstract state*, which is a set of property states, and an *execution state*.[1] Given a symbolic state $s \in S$, we denote its abstract state by $as(s)$, and its execution state by $es(s)$. The property analysis computes, for each edge in the CFG, a dataflow fact from the domain 2^S.

The pseudo code for intra-procedural property analysis is given in Figure 3. This is a standard worklist algorithm that updates a map from edges to dataflow facts, until no further updates are possible. Figure 4(a) shows the flow functions for the three types of nodes in the CFG. The flow functions first compute a set of symbolic states, and then use a grouping function $\alpha : 2^S \to 2^S$, described in detail later, to filter this set.

F_{mrg} combines the dataflow facts on its input edges into a single fact, using set union.

F_{br} takes a set of symbolic states on its input edge. It maps each input state to an output state using f_{br}, the transfer function for branch nodes. f_{br} has two effects: First, it uses a theorem prover to determine whether a given branch direction is feasible using the information in the execution state. Second, if a branch direction is neither implied by nor ruled out by the execution state, it updates the execution state, setting the branch predicate to either true or false.

F_{oth} maps input states to output states using f_{oth}, the transfer function for computation nodes. f_{oth} may update the abstract state if the node corresponds to a transition in the property FSM. It may also update the execution state.

3.1.1 *Grouping symbolic states*

Property analysis uses a function α to group together certain sets of execution states. By defining α as in Figure 4(b), we obtain three versions of property analysis corresponding

[1] $\bot$ represents no behaviors; $\top$ represents all behaviors.

to fully path-sensitive analysis, standard dataflow analysis, and property simulation:

Fully path-sensitive analysis. α_{cs} is the identity function. Therefore, merge nodes merely accumulate information from all of their predecessor edges.

Standard dataflow analysis. α_{df} merges all the symbolic states in a set into a single symbolic state. Therefore, this analysis is similar to standard dataflow analysis, in which sets of tuples are joined into a single tuple at merge points. Because of this merging, the correlation between a given property state and the execution states under which the property state arises is lost. Note that this analysis is predicate aware, in the sense that the correlation between two branches can be tracked provided there are no intervening merges between them. An efficient version of the analysis can be obtained by fixing the execution states as $\top$. This would result in a path-insensitive analysis that only propagates sets of property states.

Property simulation. α_{as} takes a set of symbolic states and "groups" the elements of the set based on the property state. All of the execution states in one group are merged. For example, $\alpha_{as}(\{[\{a\}, l], [\{a\}, m], [\{b\}, n], [\{b\}, o],\}) = \{[\{a\}, l \sqcup m], [\{b\}, n \sqcup o]\}$. In other words, α_{as} is designed to maintain the correlation between a given property state and the execution states under which the property FSM is driven into that state at a given program point.

3.1.2 *Termination and complexity*

We now argue the termination and complexity of intraprocedural property simulation. Let T be the cost of one call to the flow function for execution states (f_{br} or f_{oth}), let J and Q be the cost of the join operation and the equality operation on execution states, respectively.

By the definition of α_{as}, the number of symbolic states in each dataflow fact is bounded by $|D|$, the number of states in the property FSM. During the analysis, each element of a set can become less precise repeatedly. If we assume that each element can become less precise at most H times, then the algorithm terminates. This can be guaranteed either by requiring a finite height lattice of execution states (of height H), or by using widening operators [6]. Since the set on a given edge can have at most $|D|$ elements, and each element can become less precise at most H times, each edge is relaxed at most $H|D|$ times, causing $O(H|E||D|)$ nodes to be processed, where $|E|$ is the number of edges in the CFG.

In our implementation, when a node is added to the worklist, we keep track of the property state for which the execution state has changed. This allows us to (1) evaluate f_{br} or f_{oth} only on the execution state that has changed and (2) evaluate $\sqcup$ (in α_{as}) and the equality test (on line 21 from Figure 3) only on the newly produced execution state. As a result, each time a node is processed, there is at most one equality operation, one join operation, and one call to the flow function (f_{br} or f_{oth}). Therefore, the complexity of intraprocedural property simulation is $O(H|E||D|(T + J + Q))$.

3.1.3 *A framework for property simulation*

In the definition of property simulation given above, the choice of the domain for execution states and the join operation on execution states have been left open. Therefore, the definition provides a framework for property simulation. Particular algorithms of varying precision and complexity can be obtained by fixing the domain and the join operation used for execution states. For instance, if the domain of execution states is chosen to be sets of symbolic stores (i.e., stores arising from symbolic evaluation), and the join operation is chosen to be set union, property simulation is identical to fully path-sensitive analysis.

Therefore, property simulation should be viewed as a technique for grouping together the execution states that imply a particular property state. This grouping allows us to control the complexity of the analysis by selecting the join operation, while avoiding any loss of precision from merging execution states associated with different property states.

3.1.4 *Instantiation to constant propagation*

The particular instantiation of property simulation we use in our work is based on constant propagation. Execution states are chosen to be stores that map program variables to values from a standard constant propagation lattice. The join operation chosen is also the standard join used in constant propagation. If a variable has different values in two stores that are joined, its value is set to $\top$. The theorem prover uses the execution state to replace variables in a predicate expression with their values, and then simplifies the expression. If the resulting expression is either `T` or `F`, the appropriate branch is eliminated. Otherwise, if the simplified expression is of the form `x == c` for some constant `c`, f_{br} updates the execution state accordingly along each branch. f_{oth} updates the execution state at assignments to variables by simplifying the right hand side expression and updating the store.

For this instantiation: (1) $H = 3V$ where V is the number of variables; (2) the cost of a single call to the theorem prover is V, and therefore $T = V$ since f_{br} calls the theorem prover; (3) join and equality each take V time so that $J = Q = V$. Therefore, the complexity of the algorithm is $O(V^2|E||D|)$.

For the rest of this paper, we limit our discussion to the constant propagation instantiation of the property simulation framework.

3.1.5 *Example*

EXAMPLE 2. Figure 5 shows the dataflow facts produced by fully path-sensitive analysis (`PSA`), standard dataflow analysis (`Dataflow`), and property simulation (`PropSim`) for the program in Figure 2, given the temporal safety property in Figure 1. We abbreviate `dump` with `d`, $[\mathtt{p} \to \mathtt{F}, \mathtt{d} \to \mathtt{T}]$ with `!pd`, `$uninit` with `$u`, `Opened` with `o`, etc.

We now describe in detail the steps taken by property simulation, pointing out the points at which property simulation differs from the other two analyses.

Step 1. The theorem prover does not have enough information in the execution state to determine the direction of branch n_1, so the symbolic state is split, propagating $[\mathtt{\$u}, \mathtt{d}]$ and $[\mathtt{\$u}, \mathtt{!d}]$ to the true and false successor edges of n_1.

Step 2. n_2 transitions the property state from `$u` to `o`, thus propagating $[\mathtt{o}, \mathtt{d}]$ to its successor.

Step 3. n_3 merges the incoming facts, keeping the execution states `d` and `!d` separate because the associated property states are different. Thus, the outgoing information is $\{[\mathtt{\$u}, \mathtt{!d}], [\mathtt{o}, \mathtt{d}]\}$. Notice that `Dataflow` loses precision here compared to `PropSim`, since it merges all the execution states into one, producing $\{[\{\mathtt{\$u}, \mathtt{o}\}, \top]\}$.

Step 4. Given either state `d` or state `!d`, the theorem prover cannot determine the direction of branch n_4, so each

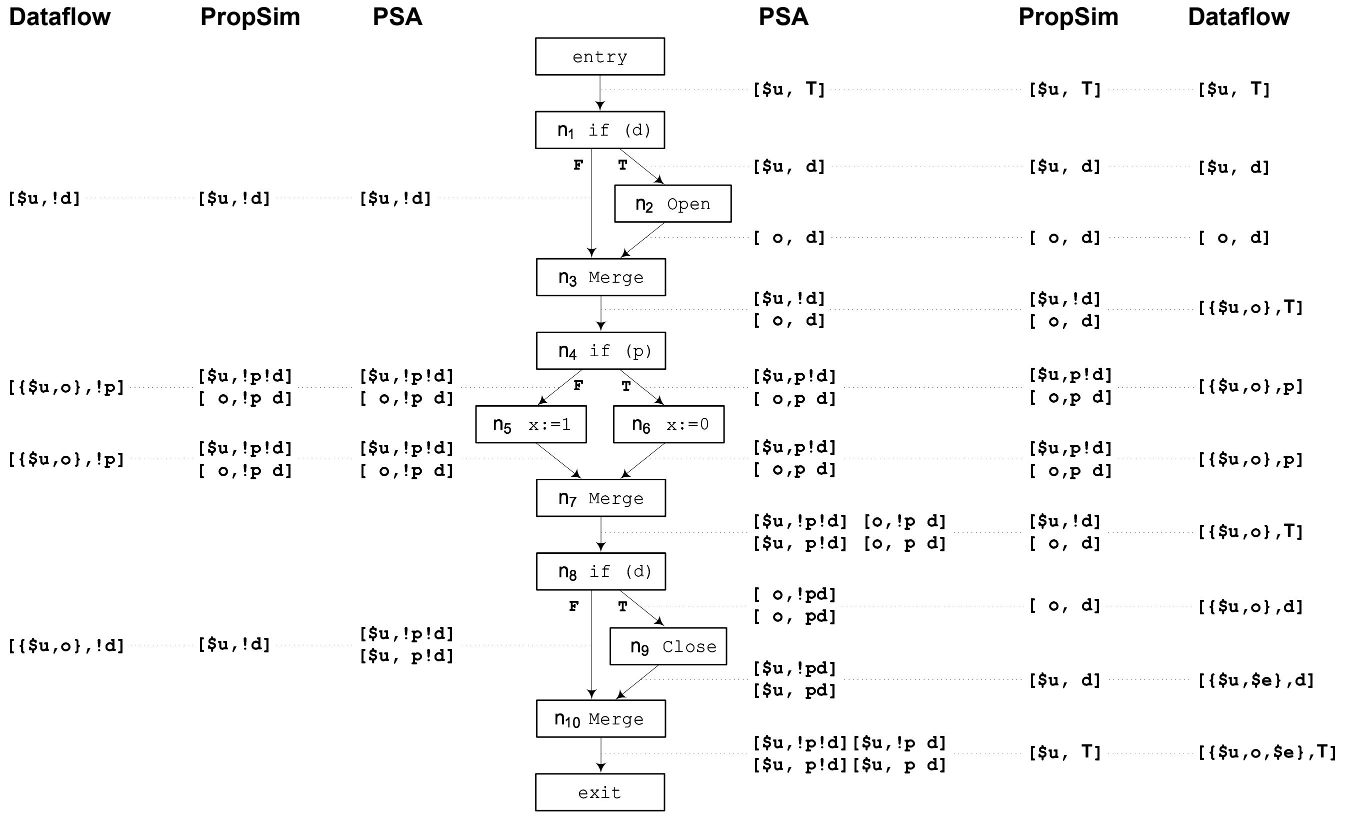

Figure 5: A comparison of fully path-sensitive analysis, standard dataflow analysis, and property simulation.

symbolic state is split, propagating $\{[\$u, p!d], [o, pd]\}$ and $\{[\$u, !p!d], [o, !pd]\}$ to the true and false successors of n_4.

Step 5-6. Nodes n_5 and n_6 add the value of `x` to the execution state. We drop this information from the execution states in order to simplify the diagram.

Step 7. n_7 merges the incoming facts $\{[\$u, p!d], [o, pd]\}$ and $\{[\$u, !p!d], [o, !pd]\}$ based on the property state. The result is $\{[\$u, !d], [o, d]\}$, which says that either the property state is `$uninit` and `dump` is false, or the property state is `Opened` and `dump` is true. This step shows how `PropSim` differs from `PSA`. In `PSA`, `p` is tracked accurately, thus doubling the number of execution states that must be analyzed downstream of n_7. However, `PropSim` drops the value of `p` from the execution state because it is not correlated with the property state.

Step 8. When the theorem prover is invoked at n_8 with either state `d` or state `!d`, it is able to determine that only one leg of the branch is feasible: true for state `d` and false for state `!d`. As a result, $\{[o, d]\}$ and $\{[\$u, !d]\}$ are propagated respectively to the true and false successors of n_8.

Step 9. n_9 transitions the property from `o` to `$u`, thus propagating $[\$u, d]$. Notice how, because of the merge at node n_3, `Dataflow` produces a transition to `$e` at n_9. □

3.2 Inter-procedural property simulation

The pseudo code for inter-procedural property simulation is given in Figure 6. Property simulation is extended to the inter-procedural case in a context-sensitive manner through the use of partial transfer functions, or summary edges, as in [27, 23]. The main idea is as follows: Suppose we are processing function `foo` and we encounter a call to function `bar`. We would like to apply a transfer function on the edge from the call site in `foo` to the associated return node in `foo`. However, since the body of `bar` contains multiple nodes, the transfer function must be generated dynamically by analyzing `bar`. This is done by maintaining and updating a summary for `bar` that maps dataflow facts at entry to `bar` to dataflow facts at exit from `bar`. When a call site is encountered in `foo` with dataflow fact `s`, the current summary for `bar` is consulted. If the summary contains an entry $s \rightarrow s'$, s' is added at the return node in `foo`. Otherwise, dataflow is triggered at the entry node of `bar`. The map *Info* is modified to associate each dataflow fact generated in the body of `bar` with the dataflow fact at entry to `bar` for which dataflow was triggered. This modification enables generation of the context-sensitive summaries described above.

Context-sensitive property simulation may not terminate if the domain of execution states is infinite, as in constant propagation. Therefore, we restrict context-sensitivity to property states. We treat execution states in a context-insensitive manner, by merging execution states from different call sites at function entry nodes using α_{as} (line 36).

The algorithm in Figure 6 can be implemented efficiently using the framework of Reps, Horwitz and Sagiv (RHS, [23]). The complexity of this implementation of property simulation is $O(V^2|D|(|E||D| + Calls|D|^2))$, where *Calls* is the number of call sites in the program. A detailed discussion of inter-procedural property simulation, its RHS implementation and its complexity is given in [8].

```
   global
1    $Worklist : 2^{N \times D}$
2    $Info : (E \times D) \rightarrow 2^S$
3    $Summary : (F \times D) \rightarrow 2^S$

   procedure Solve(Global CFG = $[N, E, F]$)
   begin
4    for each $[f, d] \in F \times E$ do $Summary(f, d) := \emptyset$
5    for each $[e, d] \in E \times D$ do $Info(e, d) := \emptyset$
6    $e := Out_T(entryNode(main))$
7    $Info(e, \$uninit) := \{[\$uninit, \top]\}$
8    $Worklist := \{[e, \$uninit]\}$

9    while $Worklist \neq \emptyset$ do
10     Remove a pair $[n, d]$ from $Worklist$
11     switch($n$)
12       case $n \in Call$:
13         $ss_{in} := Info(In_0(n), d)$
14         $ss_{out} := \emptyset$
15         for each $d' \in D$ s.t. $ss_{in}[d'] \neq \emptyset$ do
16           if $Summary(callee(n), d') \neq \emptyset$ then
17             $ss_{out} := ss_{out} \cup Summary(callee(n), d')$
18           AddTrigger($entryNode(callee(n)), d', ss_{in}[d']$)
19         Add($Out_T(n), d, \alpha_{as}(ss_{out})$)
20       case $n \in Exit$:
21         $ss_{in} := Info(In_0(n), d)$
22         AddToSummary($n, d, ss_{in}$)
23       case $n \in Merge$:
24         $ss_{out} := F_{mrg}(n, Info(In_0(n), d), Info(In_1(n), d))$
25         Add($Out_T(n), d, ss_{out}$)
26       case $n \in Branch$:
27         $ss_T := F_{br}(n, Info(In_0(n), d), true)$
28         $ss_F := F_{br}(n, Info(In_0(n), d), false)$
29         Add($Out_T(n), d, ss_T$)
30         Add($Out_F(n), d, ss_F$)
31       case $n \in Other$:
32         $ss_{out} := F_{oth}(n, Info(In_0(n), d))$
33         Add($Out_T(n), d, ss_{out}$)

34   return $Info$
   end

   procedure AddTrigger(n, d, ss)
   begin
35   $e := Out_T(n)$
36   $ss' := \alpha_{as}(ss \cup Info(e, d))$
37   Add($e, d, ss'$)
   end

   procedure Add(e, d, ss)
   begin
38   if $Info(e, d) \neq ss$ then
39     $Info(e, d) := ss$
40     $Worklist := Worklist \cup \{[dst(e), d]\}$
   end

   procedure AddToSummary(n, d, ss)
   begin
41   if $Summary(fn(n), d) \neq ss$ then
42     $Summary(fn(n), d) := ss$
43     for each $m \in returnSites(n)$ do
44       for each $d' \in D$ s.t. $Summary(fn(m), d') \neq \emptyset$ do
45         $Worklist := Worklist \cup \{[callSite(m), d']\}$
   end
```

fn: maps a node to the name of its enclosing function
entryNode: maps a function name to its entry node
callee: maps a call node to the name of the called function
callSite: maps a return-site node to its call-site node
returnSites: maps an exit node to its return-site nodes

Figure 6: Inter-procedural property simulation.

3.3 Why is property simulation precise?

Property simulation selectively merges away information from execution states. Therefore, it is possible to construct programs for which property simulation is less precise than full simulation. One such example program is given in Figure 7(a). In this program, because neither branch of the first conditional changes the property state, the correlation between `dump` and `flag` is lost. As a result, the analysis is unable to detect that two of the four paths through the remaining conditionals are infeasible. This leads to a false error report along the path in which a call to `fclose` is not preceded by a call to `fopen`.

It may appear that the program in Figure 7(a) represents a common situation in programs, namely that conditions guarding transitions are copied around. In fact, this example represents a much narrower class of programs, in which flags are copied around before they are ever used to guard transitions, and different copies of the flag are used to guard different transitions. The example programs in Figures 7(b) and 7(c) represent variations of the example in Figure 7(a) that occur much more frequently in practice.

The program in Figure 7(b) represents a situation in which the guarding flag is passed to another function, so that the guard on subsequent transitions is a different variable. Property simulation is accurate in this case. For each property state at the end of the first conditional, the value of `dump` is known. As a result, only one path is taken through the second conditional, and so the value of `flag` is known. In other words, the correlation between the property state and `flag` is preserved.

The program in Figure 7(c) represents a situation in which the guard expression is a complex, expensive operation that the programmer does not wish to repeat or cannot replicate later in the program. Therefore, a flag is set to indicate whether the file was opened; this flag is used later to decide if the file should be closed. Property simulation is accurate in this case as well, because the correlation between `flag` and the property state is preserved. This example also represents a situation in which a helper function is called to perform some transition. The helper function returns a status code indicating whether it was able to perform the transition. The flag `dump` may represent some complex condition on the state of the operating system, for instance. Because subsequent transitions are only guarded by `flag`, there is no loss of precision if the analysis is unable to maintain the value of `dump` in the execution states.

Property simulation is precise because it is designed to match the behavior of a careful programmer. In order to avoid programming errors, she must maintain an implicit correlation between a given property state and the execution states under which the property FSM is in that state. Property simulation makes this correlation explicit.

4. ESP

In the previous section, we assumed that programs update the state of a single global state machine. In practice, there are usually multiple values of a given type (for instance, file handles) created during execution of the program; each such value has an associated property FSM, the state of which changes as the program executes. In this section, we describe ESP (Error Detection via Scalable Program Analysis), a system that uses a combination of scalable alias anal-

```
if (dump)
   flag = 1;
else
   flag = 0;

if (dump)
   f = fopen(...);

if (flag)
   fclose(f);

  (a)
```

```
if (dump)
   f = fopen(...);

if (dump)
   flag = 1;
else
   flag = 0;

if (flag)
   fclose(f);

  (b)
```

```
if (dump) {
   f = fopen(...);
   flag = 1;
}
else
   flag = 0;

if (flag)
   fclose(f);

  (c)
```

Figure 7: Examples highlighting the precision of property simulation.

ysis and property simulation to track the states of multiple stateful values in large C programs.

ESP is a partial verification method that borrows a key insight from the Metal checking system [11]. The insight behind Metal is that the abstraction gap between a temporal safety property and C source code could be bridged by a programmer-supplied specification. The specification includes an FSM that encodes the property to be checked, a set of source code patterns that indicate how fragments of source code map to transitions in the property FSM, and a set of patterns that indicate how fresh stateful values are created by the program.

Figures 1 and 8 show the specification we use for verifying calls to `fprintf` in gcc. According to this specification, stateful values are created by calls to `fopen`. Each stateful value goes through state transitions during program execution. A function call that matches a pattern causes a transition on the property FSM of the value held by the expression in position `e` at the call site. If the property FSM of any stateful value reaches the error state, the program violates the safety property.

ESP is a conservative system; when it does not report any errors, the programmer is guaranteed that the specified property is not violated by the program. This characteristic imposes a heavy burden on ESP; the analysis must at once be conservative, precise and scalable.

4.1 Insights behind scalable verification

The complication introduced by multiple stateful values is that these values may flow through assignments to function calls in the source code at which the expression in position `e` appears syntactically different from the value. Therefore, a conservative system such as ESP must perform a global value flow analysis as part of the property analysis, in order to determine which stateful values are affected by a given function call in the code. However, value flow analysis based on global dataflow analysis has so far proven intractable on large programs.

The first insight behind ESP is that property analysis for multiple values can be broken up into two sub-problems, each of which can be solved by an analysis at a different precision level. We can use a highly scalable flow-insensitive but context-sensitive approximation of value flow to discover which stateful values are affected at every function call in the program that matches a pattern. We can then run property simulation on all of the stateful values, using the results of the previous analysis instead of tracking value flow directly during property simulation.

C code pattern	Transition	Creation?
`e = fopen(_)`	`Open`	Yes
`fclose(e)`	`Close`	No
`fprintf(e,_)`	`Print`	No

Figure 8: Source code patterns in ESP.

The second insight behind ESP is that by restricting the specification language to preclude properties that correlate the states of multiple values, we can analyze one stateful value at a time, through the whole program, much like a bit-vector dataflow analysis. This approach amplifies the scaling effect of the heuristic used in property simulation, because when ESP is tracking one stateful value, branches that affect the states of values other than the tracked value are merged away.

4.2 ESP analysis

In this subsection, we describe the analysis components of ESP. Our goal here is to provide enough of an overview so that it is clear how ESP utilizes property simulation. We use the simplified version of the core gcc compiler code in Figure 9 as a running example.

ESP consists of the following analysis steps.

a. CFG construction. We run a scalable points-to analysis [7] to produce a conservative approximation of the call graph of the program. We replace every indirect call with direct calls to all possible target functions at the call site. The graph is potentially quadratic in the number of call edges, but is linear in practice because of a simple caching technique explained in [7]. While constructing the call graph, we also produce CFGs for all functions.

b. Value flow computation. We run a scalable context-sensitive flow-insensitive points-to analysis [9] to produce a conservative approximation of the flow of values in the program. The analysis maps every expression to a node in a "value flow graph" (VFG), and answers value flow queries conservatively: if there is some run of the program in which the value of expression `e` flows to `e′`, then there is a path of zero or more flow edges in the VFG from `e` to `e′`. Because the VFG is context-sensitive, it is able to distinguish between value flow at different call sites to the same function.

c. Abstract CFG construction. Once the VFG is produced, we use the property specification to replace calls to

```
   FILE *f1, *f2;
   int p1, p2;
1: compFile() {
     if (p1)
       f1 = fopen(...);
     if (p2)
       f2 = fopen(...);
     restOfComp();
   }
```

```
2: restOfComp() {
     if (p1)
       i: printRtl(f1);
     if (p2)
       j: printRtl(f2);
     restOfComp();
   }
3: printRtl(FILE *f) {
     fprintf(f,...);
   }
```

(a)

$$\begin{tikzcd} 1 \arrow[d] \\ 2 \arrow[d] \arrow[loop left] \\ 3 \end{tikzcd}$$

(b)

$$\begin{tikzcd} f1 \arrow[dr, "i_2"] & \\ & f \\ f2 \arrow[ur, "j_2"'] & \end{tikzcd}$$

(c)

```
printRtl ->
  inNodes: f
restOfComp ->
  inNodes: f1, f2
compFile ->
  outNodes: f1, f2
```

(d)

Figure 9: Application of ESP on gcc.

pattern functions in the CFGs with special pattern nodes. Each pattern node is parameterized by the VFG node for the expression in position `e` at the call; this node represents the possible stateful values that may have state transitions at that program point.

EXAMPLE 3. Figures 9(b) and 9(c) show the call graph and the VFG for the program in Figure 9(a). The label $\mathtt{i}_2$ on the flow edge in the VFG indicates that the value of `f1` flows to `f` due to parameter passing at call site `i`. The subscript on `i` indicates that the call site is located in the body of function 2 (`restOfComp`). This information adds a measure of flow-sensitivity to the VFG. □

d. Interface expression computation (bottom-up slicing). In order to track value flow more accurately, we split up inter-procedural value flow into smaller, more accurate flows by introducing the concept of interface expressions. The input interface expressions (inNodes) of a function `bar` include all globals, formal parameters of `bar`, and dereferences of these. At entry to `bar`, these expressions may hold stateful values that may have their state changed during execution of `bar`. The output interface expressions (outNodes) of a function `bar` include all globals, the return value of `bar`, and dereferences of these and the formal parameters of `bar`. At exit from `bar`, these expressions may hold stateful values that were created by `bar`. In large programs, these sets can be very large, due to global variables. Further, in an unsafe language such as C, the sets cannot be pruned based on declared types.

Instead, we use a bottom-up slicing procedure on the call graph of the program to prune interface expression sets: An expression `e` can be omitted from inNodes(`bar`) if there is no pattern on expression $\mathtt{e}'$ within `bar` (or functions called by `bar`) such that the value of `e` flows to $\mathtt{e}'$ between the start of `bar` and the execution of the pattern. Similarly, an expression `e` can be omitted from outNodes(`bar`) if there is no value creation pattern on expression $\mathtt{e}'$ within `bar` (or functions called by `bar`) such that the value of $\mathtt{e}'$ flows to `e` between the execution of the creation pattern and exit from `bar`. This slicing step allows us to dramatically reduce the number of interface expressions.

Mod set computation. We also use the bottom-up slicing procedure to compute mod sets for all functions. The mod sets are represented implicitly, using VFG nodes instead of sets of variables. Mod sets are used by property simulation: if a statement is encountered that is too complicated for the simulation engine to process, the analysis can proceed conservatively by invalidating the values in the execution state for those variables that are in the mod set of the statement.

Alias set computation. ESP handles multiple stateful values by creating sets of interface expressions called "alias sets". An alias set is a set of syntactic expressions that may hold the same stateful value. The alias sets to be tracked in any function `foo` are created from subsets of the inNodes of `foo`, or from subsets of the outNodes of functions called by `foo`. Alias sets for input values are created during property simulation. Alias sets for output values are created using the bottom-up slicing procedure. One way to view this step is as an escape analysis.

The result of this procedure is that for each created value in the program, there is an alias set `s` associated with the highest (in the call graph) function `foo` to which the value escapes. We trigger property analysis for each such `s` by associating a symbolic state [$`uninit`, ⊤] with `s` at the entry node of `foo`. By triggering dataflow in this way, we can discover errors that arise because a value was used before being created (for instance, a call to `fclose` was not preceded by a call to `fopen`).

EXAMPLE 4. Figure 9(d) shows the interface expressions produced by ESP for the program in Figure 9(a). The potential inNodes of `printRtl` are `f1`, `f2`, `p1`, `p2`, and `f`. A VFG query reveals that the only one of these whose value is transferred to `f` between the entry point of `printRtl` and the call to `printf` is `f`. The inNodes of `restOfComp` are `f1` and `f2`, as they both flow to inNode `f` of callee `printRtl`. There are two created values in `compFile`. They escape through outNodes `f1` and `f2`. The top level function to which the values escape is `compFile`. The associated alias sets are {`f1`} and {`f2`}. □

e. Property simulation.

- Intra-procedural case. Suppose we are tracking the state of an alias set `s`. When we encounter a pattern on $\mathtt{e}'$, we query the VFG. If some $\mathtt{e} \in \mathtt{s}$ can flow to $\mathtt{e}'$, we update the state of `s` based on the pattern. Furthermore, if some other $\mathtt{e}''$ can flow to $\mathtt{e}'$, we add an identity transition for `s`, because the transition may not have occurred on the value being tracked. This requirement is introduced because our VFG represents only "may" information.

- Inter-procedural case. Suppose we are tracking alias set `s` in `foo`, and we encounter a call to `bar`. We query the VFG to determine which inNodes of `bar s` may flow to *at this call site*. These nodes form an alias set $\mathtt{s}'$. We trigger dataflow in `bar` from $\mathtt{s}'$ in an initial state given by the state of `s` at the call site.

Suppose we are tracking alias set `s` in `bar`, and we encounter the exit node of `bar`. At this point, we have discovered a new component of the summary of `bar`, namely a state transition $d_1 \rightarrow d_2$ for alias set `s`. When such a transition is discovered, we reflect it back to every call site c to `bar` by adding transitions $d_1 \rightarrow d_2$ at c for all alias sets in the caller that may flow to `s` at c. As in the intra-procedural case, if multiple expressions from the caller may flow to `s` at c, we add the identity transition at c.

Example 5. For the program in Figure 9(a), property simulation tracks the states of alias sets {`f1`} and {`f2`} from the entry node of `compFile`. Consider tracking {`f1`}. The first branch in `compFile` causes a split in the state of {`f1`}, so no merge is performed at the join point. The second branch does not affect the state of {`f1`}, so states are merged. As a result, information about `p2` is lost. Simulation is triggered in `restOfComp` in two ways: {`f1`} in state `Opened` with `p1=T`, and {`f1`} in state `$uninit` with `p1=F`. The first case verifies trivially because `fprintf` leaves the state `Opened` unchanged. In state `$uninit`, `p1=F` and therefore call site `i` is not reached. `p2` is not tracked in the execution states; hence, call site `j` is considered. The VFG is queried to determine the alias set in `printRtl`. {`f1`} does not flow to any inNode of `printRtl` at call site `j`, resulting in an empty alias set. Therefore, simulation is not triggered in `printRtl`. A similar argument holds for {`f2`}. □

5. CASE STUDY: FILE OUTPUT IN GCC

The goal of ESP is to verify safety properties of commercial programs, which are typically both large and written in C++. As a first step, we have built a system for analyzing large C programs. In order to understand both the efficiency and the accuracy of property simulation, we have applied ESP to the problem of verifying calls to `fprintf` in the gcc compiler. In this section, we discuss our results.

5.1 gcc

We analyze the source code of a version of the gcc compiler taken from the SPEC95 benchmark suite. The structure of this version of gcc is as follows: The main function contains a loop that invokes `compile_file` to compile individual compilation units. `compile_file` conditionally opens 15 output files based on user flags, runs compilation, and then conditionally closes the output files. At various points during the compilation process, `fprintf` is called to write out parts of the RTL and/or debugging messages to various output files, if the corresponding user flags are set and/or the files are non-`NULL`. Figure 9(a) shows a snippet of the core gcc code that summarizes its file output behavior.

gcc is a complex program: It has 140,000 LOC in 2149 functions spread over 66 files; there are 1,086 global and static variables; the call graph contains a strongly connected component (SCC) with over 450 functions. This SCC comes about because of the recursive descent nature of gcc. The complexity of the call graph is not the result of our conservative points-to analysis. Even if we ignore function pointers, this SCC has 350 functions. Worse yet, most of the calls to `fprintf` are buried in functions that either are in this SCC or are called from functions in this SCC.

5.2 Verification

File output in gcc is a difficult property for path-sensitive methods, because the code starts by conditionally creating 15 file handles based on un-correlated user flags, and all of these file handles can reach the calls to `fprintf` buried in the code. Therefore, path-sensitive methods may track all 2^{15} combinations of file states through the code representing the core engine of gcc; this would likely be infeasible. Our property simulation method, on the other hand, is ideally suited for the problem. The file handles are processed one at a time. When file i is processed, all of the conditionals containing calls to `fopen` except the i^{th} conditional are merged, because they do not affect the state of file i. Therefore, the core code of gcc is simulated in two configurations, one in which file i is open and user flag i is true, and another in which file i is not open but user flag i, which guards the calls to `fprintf` reachable by file i, is false.

We used ESP to check gcc against the specification in Figures 1 and 8. We were able to verify that there is no execution of the program in which `fprintf` is called on a file handle in states `$uninit` or `Closed`.

In addition, a VFG query confirms that the only values on which `fprintf` is called are the file handles created in `compile_file`, `stdout`, and `NULL`. The simulation guarantees that `fprintf` is never called on `NULL`, for the following reason. The VFG shows that the only way `NULL` may flow to `fprintf` is through the global file variables tracked by ESP. Suppose that `NULL` does flow to a call to `fprintf` through one of the global file variables `f`. Then when `f` is processed, this call to `fprintf` will be reached with `f` in state `$uninit`, resulting in a transition to the error state.

Put together, the conditions above guarantee that all of the 646 calls to `fprintf` in gcc print to valid, open files. This is a useful, non-trivial property that cannot be expressed using types. To our knowledge, ESP is the first program verification method to have successfully verified temporal safety properties of a program the size of gcc.

5.2.1 Methodology

We perform context-sensitive inter-procedural property simulation of the downwards closure (~140,000 LOC) of `compile_file` in the call graph of gcc. In order to successfully verify gcc, we made a few changes to the gcc source code. These changes are listed below:

- The `obstack` free function contains an indirect call to a free method that pollutes our call graph. We hand modeled this function.

- As noted in Section 3.2, our inter-procedural property simulation algorithm is context-insensitive with respect to the execution state. There are two pairs of file handles (`sched_dump_file`/`sched2_dump_file`, and `cse_dump_file`/`cse2_dump_file`) for which this merging blocks verification. For this study, we avoided the problem by creating two copies each of functions `cse_main`, `schedule_insns`, and `schedule_block`.

	RunTime (secs)	MemUsage (MB)
Average	72.9	49.7
Maximum	170	102

Table 1: Performance of property simulation.

We are exploring ways of introducing context-sensitivity with respect to execution states in a controlled manner.

5.2.2 *Why is ESP precise?*

It is easy to see that property simulation should be accurate enough to track the correlated file output behavior in gcc. Because of the merge heuristic, all of the many thousand conditionals in the code that are expensive or difficult to track are merged away with no loss of precision.

The other aspect of ESP is the value flow analysis. Our VFG is precise with respect to top level pointers, but conservative with respect to pointers stored in data structures. Because gcc does not store file handles in heap-allocated data structures, we are able to generate alias sets and track value flow accurately. We accept this as a fair advantage, because our goal in this study is to determine the usefulness of property simulation rather than the power of ESP.

In our preliminary experiments applying ESP to parts of a large commercial operating system, we have yet to find any instances in which property simulation (in particular, the constant propagation instantiation of property simulation) is inaccurate. We have not yet applied ESP in any context in which the value flow is too complicated for our value flow analysis. It is likely that as we apply ESP to check other properties, the primary precision bottleneck will be the value flow analysis. In order to address this problem, we have designed a scalable path-sensitive value flow analysis that tracks alias sets efficiently, using the merge heuristic built into property simulation.

5.3 Scalability

We have implemented inter-procedural property simulation as an extension of a global RHS-style dataflow engine [23, 8]. Some performance statistics are summarized in Table 1. The table reports average and maximum values, over the 15 file handles, of running time in seconds (wall clock), and memory usage in MB per file handle. These numbers do not include the cost of building and loading CFGs. This data was generated on a Toshiba Tecra 8200 laptop with a 1GHz Pentium III processor and 512MB RAM, running Windows XP.

The algorithm is surprisingly efficient, especially considering that we have not put significant effort into optimizing our implementation. We believe that the cost of simulation can be amortized significantly by caching states and reusing them across simulations, and by using standard ordering and representation techniques from dataflow analysis.

While we have considered only one program in our case study, we believe that the size and complexity of this program provides a compelling argument that property simulation is a scalable method for verifying temporal safety properties of large programs.

6. RELATED WORK

In this section, we survey related work on partial program verification and path-sensitive dataflow analysis.

6.1 Partial program verification

Program verification [17, 22] has been viewed as the holy grail of software reliability for decades. Over the years, it has been accepted that full scale verification of large code bases is infeasible. However, there has been a resurgence of research in recent years on partial verification: checking a program against a specification of a particular temporal safety property. We list some examples of projects along these lines below. Our work follows the lead of these projects.

Typestate analysis. Most recent work on partial verification can be viewed as tracking typestate, first introduced by Strom and Yemini in [25]. Typestate extends the ordinary types in the program, which remain invariant through the lifetime of an object, with a set of states between which values of a given type can transition. ESP can be viewed as a typestate checker for large programs.

Partial verification tools. All of these tools can statically guarantee the absence of errors.

ESC-Java [14] applies theorem proving methods to verify functions in Java programs against specifications of their pre-conditions and post-conditions. ESC-Java is a local analysis that requires programmer annotations; recent work has focused on automatically generating annotations [15]. Our method is less precise that ESC, but is global and free of annotations. The summaries produced by ESP can be viewed as automatically generated annotations.

SLAM [3] is a global path-sensitive verification method based on iterative refinement. It starts with a coarse approximation of the program and adds knowledge to the abstraction in a goal directed manner. However, if the initial abstraction is too coarse, the iteration effort may be too high. Property simulation can be viewed as a way of providing an effective starting point for the iteration in SLAM.

The flow-sensitive type qualifier system [19] is similar to the analysis engine in ESP, but is less precise, because it does not handle branch correlation, and it is not context-sensitive with respect to the property state.

Error detection tools. All of these tools statically examine only some execution paths through the code and/or ignore aliasing and complex value flow. Therefore, they cannot guarantee the absence of errors.

Metal [11] is the error detection tool closest in spirit to ESP. The Metal mechanism for specifying safety properties and bridging the gap from source code to abstract specification was the inspiration for our work. However, our goal is to guarantee the absence of errors. Unlike Metal, therefore, we use a combination of two conservative analyses.

The core engine of Metal is a local dataflow analysis similar to the standard dataflow analysis described in Section 3. In contrast, our analysis is both inter-procedural and path-sensitive[2].

PREfix [5] is a symbolic evaluator that limits exploration of program paths by truncating the search within a function after an a priori bound. PREfix has found many thousands of errors in very large code bases.

LCLint [13] is based on a mix of type system extensions

[2]Recent extensions to Metal incorporate some degree of inter-procedural analysis and path-sensitivity [16].

and dataflow analysis. This tool is essentially a local analysis that uses function annotations if provided.

Language mechanisms. The Vault programming language [10] extends the C type system with primitives for tracking the typestate of data items in the program. The programmer places annotations on function types; the compiler then performs local type checking to verify the code. The advantage of this approach is that the programmer is forced to work with the compiler to eliminate all type errors. However, the expressiveness of the language may be restricted.

Recently, the notion of typestate has been extended to roles [21]. Roles generalize typestate; the "role" of an object captures its typestate as well as its involvement in aliasing relationships. It is not yet clear whether roles can be checked in a scalable manner.

Specification generators. All of the work described above requires some specification of a property of interest from the programmer. Ammons *et. al.* describe a technique for inferring property FSMs from execution traces, by examining patterns of interactions with a library of interest [1]. Engler *et. al.* describe a static method that identifies pairs of functions that must be used in a matched fashion by looking for pairs of matched function calls along execution paths [12]. Both of these techniques can be used to generate property specifications for ESP.

6.2 Path-sensitive dataflow analysis

Our property simulation algorithm follows a long line of work on path-sensitive dataflow analysis. Previous work in this area has focused on moving from the maximal-fixed-point (MFP) solution to the meet-over-all-paths (MOP) solution; For instance, Bodik and Anik introduce a value renaming scheme in order to obtain MOP information using an MFP analysis [4]. Our particular interest is in ruling out infeasible paths.

One way of viewing our work is that we have improved the precision of dataflow analysis for our application of property analysis by tracking a finite set of predicates (those that arise from property-related branches in the code) in addition to the standard set of dataflow facts (the FSM states). Holley and Rosen describe a general method for sharpening dataflow analysis in which dataflow precision is improved by adding a finite set of predicates [18]. Tu and Padua propose a generalized SSA scheme in which SSA merge nodes are controlled by boolean predicates [26]. Ammons and Larus present a framework for improving dataflow analysis by splitting out a finite set of interesting paths from the CFG [2]. Property simulation could be viewed as an instance of these frameworks; the novelty of our approach lies in the particular choice of predicates and the precision and efficiency of the resulting analyis.

Our work can also be viewed as providing a way of performing MOP dataflow on an infinite domain of dataflow facts. Essentially, we have split the infinite domain of concrete stores into two components, one of which (the FSM states) is known to be finite. We then merge based on the finite component, guaranteeing termination of the analysis. Steffen presents a method for splitting the CFG during dataflow analysis whenever the dataflow facts along incoming edges are different [24]. If the dataflow domain is finite, this strategy leads to an MOP solution in finite time with no loss in precision. Knoop et al extended this work to infinite domains by devising a k-limiting heuristic for merging that guarantees termination [20].

Property simulation is novel in that we employ a heuristic for termination that matches our intuition about verification: we maintain precision for those branches that appear to affect the property to be checked. Our results show that this heuristic employs the correct merge strategy at join points. Our work is also different in that we exploit the special properties of the finite component of the domain to obtain a polynomial complexity bound for our algorithm. This is important because our technique is intended for application on large programs.

We have implemented inter-procedural property simulation as an extension of the context-sensitive algorithm of Reps, Horwitz and Sagiv [23]. This formulation allows us to obtain a more efficient inter-procedural algorithm, and better argue complexity.

7. CONCLUSIONS

The application of verification technology to large programs has long been a desirable but unachievable goal. In this paper, we have described a new algorithm for path-sensitive program verification that may offer a way of approaching this goal. Our property simulation algorithm runs in polynomial time and space and is designed to capture the common case of correlated branch behavior in programs. We have used the algorithm to provide the first verification of temporal safety properties for a program of the size of gcc.

Acknowledgements

We would like to thank the anonymous reviewers for their comments, Stephen Adams for implementing aspects of ESP, Matthai Phillipose for his help in re-formulating the algorithm, Jim Larus for reviewing drafts of this paper, members of the PPRC team at Microsoft Research for building the parsing infrastructure used by ESP, and members of the Software Productivity Tools group at Microsoft Research for their help in designing ESP.

REFERENCES

[1] G. Ammons, R. Bodik, and J. Larus. Mining specifications. In *Conference Record of the Twenty-Ninth ACM Symposium on Principles of Programming Languages*, 2002.

[2] G. Ammons and J. Larus. Improving data-flow analysis with path profiles. In *Proceedings of the ACM SIGPLAN 98 Conference on Programming Language Design and Implementation*, 1998.

[3] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of SPIN '01, 8th Annual SPIN Workshop on Model Checking of Software*, May 2001.

[4] R. Bodik and S. Anik. Path-sensitive value-flow analysis. In *Symposium on Principles of Programming Languages*, pages 237–251, 1998.

[5] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. *Software - Practice and Experience*, 30(7):775–802, 2000.

[6] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, 1977.

[7] M. Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, 2000.

[8] M. Das, S. Lerner, and M. Seigle. ESP: Path-Sensitive Program Verification in Polynomial Time. Technical Report MSR-TR-2002-41, Microsoft Corporation, 2002.

[9] M. Das, B. Liblit, M. Fähndrich, and J. Rehof. Estimating the Impact of Scalable Pointer Analysis on Optimization. In *8th International Symposium on Static Analysis*, 2001.

[10] R. Deline and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, 2001.

[11] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the sixth USENIX Conference on Operating systems design and implementation*, 2000.

[12] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, 2001.

[13] D. Evans. Static detection of dynamic memory errors. In *Proceedings of the ACM SIGPLAN 96 Conference on Programming Language Design and Implementation*, 1996.

[14] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 2002.

[15] C. Flanagan and R. Leino. Houdini, an annotation assistant for esc/java. In *Symposium of Formal Methods Europe, March 2001.*, 2001.

[16] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 2002.

[17] C. A. R. Hoare. An axiomatic basis for computer programming. In *C. A. R. Hoare and C. B. Jones (Ed.), Essays in Computing Science, Prentice Hall.* 1989.

[18] L. Holley and B. Rosen. Qualified dataflow analysis. In *Conference Record of the Seventh ACM Symposium on Principles of Programming Languages*, 1980.

[19] A. Aiken J. S. Foster, T. Terauchi. Flow-Sensitive Type Qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 2002.

[20] J. Knoop, O. Rüthing, and B. Steffen. Expansion-based removal of semantic partial redundancies. In *Proceedings of the 8th International Conference on Compiler Construction (CC'99) (Amsterdam, The Netherlands)*, Lecture Notes in Computer Science, vol. 1575, pages 91 – 106. Springer-Verlag, Heidelberg, Germany, 1999.

[21] V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *Conference Record of the Twenty-Ninth ACM Symposium on Principles of Programming Languages*, 2002.

[22] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *TOPLAS: ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[23] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural data ow analysis via graph reachability. In *Conference Record of the Twenty-Second ACM Symposium on Principles of Programming Languages*, 1995.

[24] B. Steffen. Property-oriented expansion. In *LNCS 1145, 3rd International Symposium on Static Analysis, 1996*, pages 22–41. Springer-Verlag, 1996.

[25] R. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171, 1986.

[26] P. Tu and D. Padua. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In *Proceedings of the 1995 ACM International Conference on Supercomputing*, Barcelona, Spain, 1995.

[27] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN 95 Conference on Programming Language Design and Implementation*, 1995.