Joining Dataflow with Predicates

Jeffrey Fischer
UC Los Angeles
fischer@cs.ucla.edu

Ranjit Jhala
UC San Diego
jhala@cs.ucsd.edu

Rupak Majumdar
UC Los Angeles
rupak@cs.ucla.edu

ABSTRACT

Dataflow analyses sacrifice path-sensitivity for efficiency and lead to false positives when used for verification. Predicate refinement based model checking methods are path-sensitive but must perform many expensive iterations to find all the relevant facts about a program, not all of which are naturally expressed and analyzed using predicates. We show how to join these complementary techniques to obtain efficient and precise versions of any lattice-based dataflow analysis using *predicated lattices*. A predicated lattice partitions the program state according to a set of predicates and tracks a lattice element for each partition. The resulting dataflow analysis is more precise than the eager dataflow analysis without the predicates. In addition, we automatically infer predicates to rule out imprecisions. The result is a dataflow analysis that can adaptively refine its precision. We then instantiate this generic framework using a *symbolic execution lattice*, which tracks pointer and value information precisely. We give experimental evidence that our combined analysis is both more precise than the eager analysis in that it is sensitive enough to prove various properties, as well as much faster than the lazy analysis, as many relevant facts are eagerly computed, thus reducing the number of iterations. This results in an order of magnitude improvement in the running times from a purely lazy analysis.

Categories and Subject Descriptors: D.2.4 [Software Engineering]: Software/Program Verification; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs.

General Terms: Languages, Verification, Reliability.

Keywords: Dataflow analysis, model checking, predicate abstraction, counterexample analysis.

1. INTRODUCTION

Dataflow analysis is an efficient and heavily used technique for static program analysis. It has been applied extensively for compiler optimizations and more recently, to verify properties of software. Dataflow analyses can be classified as *eager* or *lazy* as follows.

Classical dataflow analyses [22, 11] are *eager*: they start with a fixed set of facts, and obtain the least solution to a system of equations which determines which facts are guaranteed to hold at various program points. In order to be efficient, such analyses sacrifice precision in two ways. First, by ignoring correlated branches, dataflow facts are propagated across infeasible program paths. Second, by merging at join points, dataflow facts from different paths are unified, leading to further over-approximation. In the context of verification, this imprecision is manifested as *false positives*, where owing to its over-conservative nature, the analysis is unable to prove the property of interest. Often, the very large number of false positives overwhelms the programmer and thus renders the analysis useless.

In contrast, recent work has looked at *lazy* analyses wherein there is a specific property that is to be verified. The lazy analysis proceeds by using a very simple and coarse lattice, which is then iteratively refined using false positives until the property of interest is either proved or disproved [3, 9, 5, 20, 8]. Lattices based on *predicate abstraction* [1, 18] are particularly suited to lazy analyses. Here, the dataflow facts correspond to predicates over program variables. By using relevant predicates, the analysis can soundly prune away infeasible paths as well as avoid joining along paths that must be analyzed in isolation. The lattice is refined by adding predicates that rule out false positives arising from the analysis using the smaller (coarser) set of predicates [19].

There are two difficulties with the lazy, predicate refinement based analyses. First, all the relevant information about the program must be expressed as, and reasoned about, using predicates over program variables. While this framework is very expressive in theory, in practice the analysis is restricted to predicates that automatic decision procedures can reason about efficiently. For example, it is cumbersome and inefficient to encode information about heap structures using predicates, and in these settings, other representations are more amenable to easy manipulation. Second, all the information that is relevant to the property must be found through several iterations, each of which eliminates some false positives. This is expensive due to the number of iterations, and due to the cost of the procedure by which refinement is done. A well chosen eager analysis, on the other hand, would be be able to deduce many of the relevant facts, in a single pass.

The first contribution of this paper is to show how to combine the complementary strengths of the lazy and eager

ESEC-FSE'05, September 5–9, 2005, Lisbon, Portugal.

approaches to obtain efficient, path sensitive dataflow analyses. The main technical innovation is the use of *predication* to refine any lattice-based dataflow analysis.

For any lattice $\mathcal{L}$ and a set of predicates P, the elements of the predicated lattice $\mathcal{L}_P$ are maps from P to $\mathcal{L}$. Intuitively, the predicated lattice partitions the set of program states using the predicates, and tracks a different lattice element (from $\mathcal{L}$) for each individual partition (predicate). The transfer function for the predicated lattice is computed automatically from the transfer function of $\mathcal{L}$ and the transfer function for the set of predicates [18]. The operator used to combine the dataflow facts at join points is a *predicated join*: given two maps, the predicated join maps each predicate p to the $\mathcal{L}$-join of the arguments' images of p. Thus, if along two paths different predicates hold, the result of the predicated join is that the $\mathcal{L}$-elements resulting from the two paths are separately tracked, giving a more precise result. In addition, by using predicates, the analysis is able to avoid the loss of precision that arises from propagating dataflow information over infeasible paths.

Our analysis begins with the user supplied dataflow lattice and any initial set of predicates. As in lazy schemes, the set of predicates is refined using false positives until the property of interest can be proved or disproved, or until the residual false positives are small. The refinement algorithm is a modification of the counterexample refinement step in software model checking [19] that adds predicates which either rule out infeasible paths or keep the lattice elements separate at a join point. As a result, we are able to obtain a path-sensitive version of *any* eager, lattice-based dataflow analysis.

The second contribution of this paper is to demonstrate the effectiveness of predicated lattices by instantiating the above framework with a *symbolic execution lattice* that captures dataflow facts needed in the verification of typestate properties. The symbolic execution lattice tracks an abstract heap through the program execution. This enables us to precisely track typestate values through the program execution. The symbolic execution lattice precisely tracks aliasing relationships between variables. In particular, it introduces value-tagged *may-pointers* that track the values stored at a pointer dereference even though the particular pointer target is unknown. Value-tagged may-pointers are similar to the `restrict` annotation [2] and allow strong updates during the dataflow analysis to be performed automatically.

We have implemented the symbolic execution lattice and the predicated dataflow analysis algorithm in our software model checker Blast [20]. In particular, we have implemented a predicated lattice based on the symbolic execution lattice. We show the performance of this algorithm using two sets of experiments on Windows device drivers. The typestate property deals with correct handling of I/O requests, and is a finite state machine with 22 states. We show that the predicated dataflow analysis requires significantly fewer refinement steps to prove a program correct, which in turn, translates to much faster running times. For our largest benchmark, the new algorithm runs in about five minutes, whereas the lazy predicate abstraction algorithm takes almost an hour and a half. The reason for this dramatic speedup is that the lazy technique performs several iterations in order to add predicates that track certain status values needed to prove the property, while the (eager) symbolic execution lattice is able to capture all that information and thus significantly decreases the number of refinements. Second, we show that additional refinements beyond those obtained from the specification typestate automaton are necessary for this property: in particular, the algorithm of [13] would generate false positives on each of those benchmarks. Hence, we believe our approach opens the way to precise and efficient dataflow analyses that enjoy the strengths and eliminate the weaknesses of lazy and eager techniques.

Related Work. Our work is related to *qualified* dataflow analysis [21, 15, 16], where an *a priori* fixed set of qualifications are used to make dataflow analysis more precise. Restricted forms of predicates have been used as qualifiers: these have either been tailored for specific analyses [30, 29], or use restricted control flow tests from the program [6, 23]. For example, [23] only considers predicates whose values do not change over the regions they are tracked. In contrast, our refinement scheme can generate well-scoped predicates that are not syntactically present in the program [19]. Refining join points have been considered in [13], where a fixed set of specification states are used to distinguish lattice elements at join points. Their analysis lattice is very similar to our symbolic execution lattice (but they do not tag may-pointers with values). However, the analysis can produce false positives if these states are not enough to distinguish different control flow paths of the program. In fact, for the drivers we have considered in our experiments, we found that specification-based distinguishing is not precise enough to prove properties associated with the specification. Our work is a generalization of [13], where the set of predicates can be expanded as needed based on previous counterexamples, and the predicates are not fixed to be specification states.

Predicated lattices are a special case of *reduced cardinal powers* of lattices (also called *tensor product*), which is the set of all monotone functions from the first lattice to the second [12, 25]. However, abstract interpretation with a reduced cardinal power lattice does not deal with successive automatic refinements of lattices, nor does that work address implementation issues.

Counterexample-guided abstraction refinement [5, 9, 20] has been a successful paradigm to check control properties of software. In this approach, an initially coarse model of a program is checked and then made more precise based on errors found in the abstraction which cannot be realized in the original program. The main abstraction mechanism is *predicate abstraction* [18]. However, predicates are awkward to express certain properties of program state, especially heap properties. Our work presents a generalization of predicate-based software model checking to software model checking over more general abstract structures. In particular, our symbolic execution lattice can be used to merge predicate-based software model checking with shape analysis [27], thus enabling precise model checking of heap-manipulating programs.

Several tools perform precise symbolic execution on a subset of program paths [10, 7, 24, 31], checking the typestate properties along each path. While very precise, this approach is inherently unsound, since there are infinitely many execution paths, not all of which are checked. Thus, if the analysis does not find bugs, the program may or may not have bugs.

```
Example() {
    FILE *out;
    int flag;
1:  out->is_open := 1;
2:  assert(out->is_open=1);
3:  out->is_open := 0;
    ⋮
4:  if(flag)
5:      out->is_open := 1;
    ⋮
6:  ⋮
7:  if(flag)
8:      assert(out->is_open);
}
```

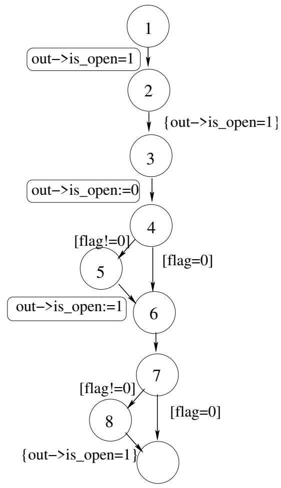

Figure 1: An program and its CFA

Organization. The rest of the paper is organized as follows. In Section 2, we recall the basic dataflow analysis framework. In Section 3, we define predicated lattices, predicated dataflow analysis algorithms, and automatic refinement schemes. In Section 4, we describe a specific *symbolic execution* lattice that can be used to precisely analyze safety properties of software. In Section 5, we describe an implementation of the algorithm and experimental results.

2. DATAFLOW ANALYSIS

We illustrate our analysis on a small imperative language with integer variables and references. For simplicity, we describe the intraprocedural analysis. Our techniques generalize to interprocedural analysis in a standard way [26].

2.1 Syntax and Semantics

Operations. In the sequel, X denotes the set of program variables. Our programs are built using operations *Ops* which are of two kinds:
(1) An *assignment* operation is of the form $\mathtt{l} := \mathtt{e}$; which assigns the value of the expression e to the variable $\mathtt{l}$,
(2) An *assume* operation is of the form $\mathtt{assume(p)}$; if the boolean expression $\mathtt{p}$ evaluates to *true*, then the program continues, and otherwise the program halts. Assumes are used to model branch conditions.

Control Flow Automata. We model a program as a *control flow automaton (CFA)* $C = (PC, pc_e, E)$, which is a rooted, directed graph with:
(1) a set of control locations (or program counters) PC which includes a special entry location $pc_e \in PC$,
(2) a set of edges $E \subseteq PC \times Ops \times PC$. We write $(pc, \mathtt{op}, pc')$ or $pc \xrightarrow{\mathtt{op}} pc'$ to denote the edge from pc to pc' labeled $\mathtt{op}$. A CFA is the control flow graph of a program; its nodes correspond to program locations, and its edges correspond to the commands that take the program from one location to the next. A *path* to a location pc is a sequence of edges $pc_e \xrightarrow{\mathtt{op}_1} pc_1 \ldots \xrightarrow{\mathtt{op}_n} pc_n$ where $pc_n = pc$.

Example 1: [**CFA**] Figure 1 shows a C function `Example`, and its CFA. The function `Example` opens and writes to a file pointer `out`. For ease of exposition, we model opening by setting the `is_open` field to 1, and we model writing via the assertion that the file must be open. After writing, the file is closed, modeled by setting the `is_open` field to 0. Then, conditionally on the variable `flag`, it opens the file pointed to by `out`. This is modeled by the if block on lines 4 and 5 that sets the field `is_open` if `flag` is true. Assume that the intervening statements in line 6 do not modify any of these variables. Finally, in lines 7 and 8, the function writes to the file pointed to by `out` conditioned on *flag*. This is modeled by the assertion on line 8 that states that only open files can be written. The CFA for `Example` is shown on the right in Figure 1. The vertices of the CFA correspond to locations of the C function. The edges are labeled by the instructions that are executed as control moves from the source to the target location. The edges labeled with boxes are assignments; those labeled with brackets correspond to assumes. We write assertions in curly braces. □

States and Transitions. A *state* is a type-preserving valuation for the variables X. Let S denote the set of all states. For a state s and variable x, let $s(x)$ denote the value of the variable x in state s. This is extended naturally to obtain values of expressions and predicates over X. Each operation $\mathtt{op}$ corresponds to a transition relation $\overset{\mathtt{op}}{\rightsquigarrow} \subseteq S \times S$ as follows. We say that $s \overset{\mathtt{op}}{\rightsquigarrow} s'$ if:

$$s' = \begin{cases} s & \text{if } \mathtt{op} \equiv \mathtt{assume(p)} \text{ and } s \models \mathtt{p} \\ s[\mathtt{l} \mapsto s.\mathtt{e}] & \text{if } \mathtt{op} \equiv \mathtt{l} := \mathtt{e} \end{cases}$$

The elements of transition relations are called *transitions*. A path $pc_o \xrightarrow{\mathtt{op}_1} pc_i \ldots \xrightarrow{\mathtt{op}_n} pc_n$ in the CFA is *feasible* if there exists a sequence of states $s_0, \ldots, s_n$ such that for all $1 \leq i \leq n$, we have $s_{i-1} \overset{\mathtt{op}_i}{\rightsquigarrow} s_i$.

We lift the transition relation to sets of states r via the strongest postcondition operator SP defined as: $\mathsf{SP}(r, \mathtt{op}) = \{s' | \exists s \in r. s \overset{\mathtt{op}}{\rightsquigarrow} s'\}$. We extend SP to sequences of operations as: $\mathsf{SP}(r, \mathtt{op}_1; \ldots; \mathtt{op}_n) = \mathsf{SP}(\mathsf{SP}(r, \mathtt{op}_1), \mathtt{op}_2; \ldots; \mathtt{op}_n)$. Finally, we define the set of *reachable states* $\mathsf{Reach}(r, pc)$ from the location pc to be the union of all $\mathsf{SP}(r, \mathtt{op}_1; \ldots; \mathtt{op}_n)$ such that there exists a path $pc_e \xrightarrow{\mathtt{op}_1} pc_1 \ldots pc_{n-1} \xrightarrow{\mathtt{op}_n} pc$ in the CFA C.

2.2 Lattice-based Dataflow Analysis

We now define a dataflow analysis framework for the CFA program representation.

Lattices. A *semi-lattice* $\mathcal{L} = (\Theta, \bot, \top, \sqcup, \sqsubseteq, \widehat{\mathsf{SP}}, [\![\cdot]\!])$ for variables X comprises a (possibly infinite) set Θ of *lattice elements*, elements $\bot$ and $\top$ of Θ, a total function $\sqcup : \Theta \times \Theta \to \Theta$ called the *join*, a preorder $\sqsubseteq \subseteq \Theta \times \Theta$, a monotone total function $\widehat{\mathsf{SP}} : \Theta \times Ops \to \Theta$ called the *transfer function*, and a total function $[\![\cdot]\!] : \Theta \to 2^S$ called the *concretization*, such that for all elements $\theta, \theta' \in \Theta$ and every label $\mathtt{op} \in Ops$, we have:

$$\begin{array}{lcl} [\![\bot]\!] = \emptyset & \text{and} & [\![\top]\!] = S \\ [\![\theta \sqcup \theta']\!] & \supseteq & [\![\theta]\!] \cup [\![\theta']\!] \\ [\![\widehat{\mathsf{SP}}(\theta, \mathtt{op})]\!] & \supseteq & \mathsf{SP}([\![\theta]\!], \mathtt{op}) \end{array}$$

A lattice element θ is an abstract representation of the set $[\![\theta]\!]$ corresponding to its concretization, and the transfer function is an over-approximation of the strongest postcondition [11].

Dataflow Analysis via Fixpoints. A dataflow analysis computes an over-approximation of the set of reachable states of a program by computing fixpoints over a lattice of abstract program states. Given a tuple $I = \langle C, \mathcal{L}\rangle$ comprising a CFA C and a lattice $\mathcal{L}$, the dataflow *problem* is to find a map D from the CFA locations PC to lattice elements such that:

$$\top = D(pc_e) \tag{1}$$

and, for each $pc \in PC$

$$\bigsqcup_{pc' \xrightarrow{\mathsf{op}} pc \in E} \widehat{\mathsf{SP}}(D(pc'), \mathsf{op}) \sqsubseteq D(pc) \tag{2}$$

We call such maps *solutions* for the dataflow problem I. There is a pointwise partial-order on solutions as follows: Let D_1 and D_2 be solutions for two (possibly different) dataflow problems $\langle C, \mathcal{L}_1\rangle$ and $\langle C, \mathcal{L}_2\rangle$. We say $D_1 \preceq D_2$ if for each $pc \in PC$, we have $[\![D_1(pc)]\!]_1 \subseteq [\![D_2(pc)]\!]_2$. For every dataflow problem, we are typically interested in the *least* solution w.r.t. $\preceq$. Such solutions are guaranteed to exist and can be found by computing the *least fixpoint* of the above set of equations [11]. The height of a lattice $\mathsf{Height}(\mathcal{L})$ is the cardinality of the largest strictly ascending chain of elements of the lattice.

Proposition 1. *For every dataflow problem $I = \langle C, \mathcal{L}\rangle$:*

1. *For every solution D, we have that for each $pc \in PC$ the set $\mathsf{Reach}([\![\top]\!], pc) \subseteq [\![D(v)]\!]$,*
2. *The least solution for I can be computed in time linear in $|E| \times \mathsf{Height}(\mathcal{L})$.*

2.3 Typestate Inference by Dataflow Analysis

To demonstrate latticed-based dataflow analysis, we apply this framework to typestate inference. Typestates [28] or flow-sensitive type qualifiers [17, 2] are used to refine types with flow-sensitive properties that may differ across program locations. As an example, we consider the type `FILE` used for I/O operations on files. Files must be used in specific ways: a file must be opened before it is accessed (read or written), and a file cannot be accessed after it is closed. To model the state of the file, we introduce typestates `closed, open`, which denote that a file is closed or open, respectively, and $\top$ denoting a file is in an unknown state. Upon allocation, a file is in a closed state. The function call `fopen` sets it to an open state, and the `fclose` sets an open file to closed. The `fprintf` function writes to an open file, and raises an error if the argument file is closed. The typestate inference problem is to find the typestates of each file at each program location. With this information, we can ascertain if, at each program point where a file is written, its typestate is `open` and if not, we know a runtime error may occur at that location.

Given a set of typestates, we define a lattice whose elements are maps from variables of type `FILE` to typestates mapping a file to its state, together with a special element $\bot$. The concretization function is defined in the standard way. The top element maps every file variable to $\top$. The join of two maps θ_1 and θ_2 is the map $\lambda x.(f(x) \sqcup_F g(x))$, where $q \sqcup_F q'$ is q if $q = q'$ and $\top$ otherwise. We have $\theta_1 \sqsubseteq \theta_2$ if for every variable $\mathtt{f}$ we have $\theta_2(\mathtt{f})$ either equals $\theta_1(\mathtt{f})$ or $\top$. Finally, $\widehat{\mathsf{SP}}(\theta, \mathsf{op})$ is (1) $\theta[\mathtt{f} \mapsto \mathtt{O}]$ if op is a call `fopen(f)`, (2) $\theta[\mathtt{f} \mapsto \mathtt{C}]$ if op is a call `fclose(f)`, and (3) θ otherwise. To solve the typestate verification problem, we compute the least solution of the dataflow problem corresponding to the above-defined typestate lattice. At any location where `f` is read or written, if the dataflow solution maps `f` to `O` (resp. `C` or $\top$), then we know a runtime error cannot (resp. may) occur at that location.

pc	Typestate	Predicated Typestate
1	$*out$:C	$true$:[$*out$:C]
2	$*out$:O	$true$:[$*out$:O]
3	“”	“”
4	$*out$:C	$true$:[$*out$:C]
5	“”	$(flag \neq 0)$:[$*out$:C]
6	$*out$:$\top$	$(flag \neq 0)$:[$*out$:O], $(flag = 0)$:[$*out$:C]
7	“”	“”
8	“””	$(flag \neq 0)$:[$*out$:O]

Figure 2: Dataflow Solutions: Locations, Typestate Lattice Solution, Predicated Typestate Lattice Solution (Each p not shown in domain maps to $\bot$)

Example 2: **[Typestate Verification]** In function `Example`, we model the function call `fopen` (resp. `fclose`) by setting the `is_open` bit of a structure FILE to 1 (resp. 0), and we model file accesses with an assertion that the passed file structure has the `is_open` bit set to 1. To verify the assertions, we compute the least solution of the dataflow problem corresponding to the CFA and the typestate lattice. The least solution at location 2 maps `out` to the typestate `O`; and hence we are guaranteed that the assertion at location 2 is never violated at runtime. Unfortunately, the solution is a conservative over-approximation of the reachable states. In our example, the least solution at location 8 maps the file `out` to the state $\top$ because there are two paths to the node 8; on one path the file `out` is in state `O` (along the “then” branch of the conditional on location 4), and on the other, `out` is in state `C` (along the “else” branch at location 4). Hence, the dataflow analysis at location 8 “joins” the values flowing in along the two paths, *i.e.,* maps `out` to $\mathtt{O} \sqcup_F \mathtt{C}$, which is $\top$, which flows to location 8. Hence, based on this analysis, we can only conclude that `out` may possibly be in the `C` state on line 8, and thus we conclude that a runtime error may occur at location 8. However, this is a *false positive*, since no actual program execution can cause the assertion to fail. The dataflow analysis loses information at locations where control-flow merges (*i.e.,* join points), and does not realize that the conditional `flag` ensures that `out` is always in state `O` at line 8. □

3. PREDICATED DATAFLOW ANALYSIS

We now describe how to obtain a more precise solution for any lattice-based dataflow analysis by using first-order predicates over the program variables to refine the information provided by the lattice.

Predicate Abstraction. Let P be a finite set of quantifier-free first-order boolean predicates over the program variables X such that the disjunction of all the predicates of P is a valid formula. The P-abstract transition relation $\rightsquigarrow_P \subseteq P \times Ops \times P$, is defined as: $p \overset{\mathsf{op}}{\rightsquigarrow}_P p'$ if there exists states s, s' satisfying predicates p, p' respectively, such that $s \overset{\mathsf{op}}{\rightsquigarrow} s'$. We say a path $pc_0 \xrightarrow{\mathsf{op}_1} pc_1 \ldots \xrightarrow{\mathsf{op}_n} pc_n$ is P-feasible if there exist $p_0, \ldots, p_n \in P$ such that for each $1 \leq i \leq n$, we have $p_{i-1} \overset{\mathsf{op}_i}{\rightsquigarrow}_P p_i$, and P-infeasible otherwise. There are standard techniques for computing $\rightsquigarrow_P$ using predicate transformers (*i.e.,* the strongest postcondition [14]) and decision procedures [18].

EXAMPLE 3: **[Predicate Abstraction]** Consider the set of predicates $P = \{flag = 0, flag \neq 0\}$. For the assume operation $\mathtt{op} = \mathtt{assume}(\mathtt{flag}! = \mathtt{0})$ corresponding to the "then" condition of the branch at location 4, we have the single P-abstract transition: $(flag \neq 0) \overset{\mathtt{op}}{\rightsquigarrow}_P (flag \neq 0)$. For the assignment operation $\mathtt{op} = \mathtt{flag} := \mathtt{1}$, we have the two P-abstract transitions: $(flag = 0) \overset{\mathtt{op}}{\rightsquigarrow}_P (flag \neq 0)$ and $(flag \neq 0) \overset{\mathtt{op}}{\rightsquigarrow}_P (flag \neq 0)$. □

3.1 Joining Lattices with Predicates

Our main technique for more precise dataflow analysis is the predicated lattice construction from a lattice and a set of predicates. For a finite set of predicates P, and a lattice $\mathcal{L} = (\Theta, \bot, \top, \sqcup, \sqsubseteq, \widehat{\mathsf{SP}}, [\![\cdot]\!])$, we define the *predicated lattice* $\mathcal{L}_P$ as the tuple $(\Theta_P, \bot_P, \top_P, \sqcup_P, \sqsubseteq_P, \widehat{\mathsf{SP}}_P, [\![\cdot]\!]_P)$ where:

1. Θ_P is the set of maps $P \rightarrow \Theta$, and $\bot_P$ is the map $\lambda p.\bot$ and $\top_P$ the map $\lambda p.\top$,
2. $\theta_1 \sqcup_P \theta_2$ is the map $\lambda p.(\theta_1(p) \sqcup \theta_2(p))$,
3. $\theta_1 \sqsubseteq_P \theta_2$ iff for all $p \in P$, we have $\theta_1(p) \sqsubseteq \theta_2(p)$,
4. $[\![\theta]\!]_P = [\![\sqcup\{\theta(p) \mid p \in P\}]\!]$, and
5. $\widehat{\mathsf{SP}}_P(\theta, \mathtt{op}) = \lambda p'.(\bigsqcup\{\widehat{\mathsf{SP}}(\theta(p), \mathtt{op}) \mid p \overset{\mathtt{op}}{\rightsquigarrow}_P p'\})$.

It is easy to check that the above satisfies the conditions on lattices, and that $\mathsf{Height}(\mathcal{L}_P)$ is $|P| \times \mathsf{Height}(\mathcal{L})$. This is an instance of a reduced cardinal power of lattices [12]. Given a predicated lattice $\mathcal{L}_P$, the dataflow problem $\langle C, \mathcal{L}_P\rangle$ can be solved by the fixpoint iteration algorithm that finds the least solution for Equations (1) and (2). The following theorem states that the resulting dataflow facts become no less precise when the set of predicates is increased.

PROPOSITION 2. *For every dataflow problem $I = \langle C, \mathcal{L}\rangle$, and set of predicates P,*

1. *the least solution D_P of $I_P = \langle C, \mathcal{L}_P\rangle$ can be computed in time linear in $|E| \times |P| \times \mathsf{Height}(\mathcal{L})$,*
2. *for every $P' \supseteq P$, the least solution $D_{P'} \preceq D_P$, and,*
3. *if P is the trivial set $\{true\}$, then D_P equals the least solution for I.*

EXAMPLE 4: **[Predicated Lattices]** Consider the program from Figure 1. We use the predicated lattice obtained from the set of predicates $P = \{flag = 0, flag \neq 0\}$, and the typestate lattice of Section 2.3 to verify the assertions in the program of Figure 1. The dataflow solution for this predicated lattice is shown in Figure 2, in the third column. Notice that at location 6, the join point, the predicated lattice solution maps the predicates $(flag = 0)$ which holds along the "else" branch to the typestate lattice element where $*out$ is still closed, and the predicate $(flag \neq 0)$ which holds along the "then" branch to the typestate lattice element where $*out$ is open. By keeping the information along the two branches separate, our analysis is precise enough to prove the assertion. In this example, we assume that the predicates are provided in advance. In the next section, we show how they can be inferred. □

3.2 Predicate Refinement

There are two sources of imprecision in a dataflow analysis. The first is that the analysis considers all program paths, instead of just those paths that are feasible. The second is the information lost at join operations. Both sources are essential: they guarantee the computability of the least solution for lattices of finite height.

Even with predicated lattices, the dataflow solution may be imprecise if the set of predicates P is inappropriate. In such cases, we would like to automatically refine the analysis to rule out the over-conservative (imprecise) solutions which yield false positives, and repeat the process until the least solution is sufficiently precise. In our case, the refinement is done by enriching the set P with new predicates and using the resulting predicated lattice.

To find the new predicates, we adapt standard predicate refinement techniques [19] to this setting, as briefly outlined below. Given a dataflow solution, if we find it is precise enough (*e.g.,* to verify a typestate property) then we are done. Otherwise, either we can find a feasible execution that violates the typestate property, or we can determine that the analysis is too imprecise. In the latter case, we determine whether the imprecision is because the analysis considered an infeasible path or because of a join. In the first case we infer new predicates P' such that path is P'-infeasible. In the second case, we find two paths that join at some location such that both paths are feasible, and the lattice element obtained at the end along either path is precise enough, but the solution obtained by joining the lattice elements at the common location leads to imprecision. In this case, we infer new predicates P' that separate the paths, *i.e.,* such that the predicates that hold along the prefixes leading to the join point are disjoint, as a result of which the subsequent predicated join does not join the lattice elements, leading to a more precise solution.

The predicated lattice construction and the refinement scheme outlined above gives us an algorithm for adaptive path sensitive dataflow analysis. The input to the algorithm is a dataflow problem $I = \langle C, \mathcal{L}\rangle$ and optionally a set of predicates P. In each step, we compute the least fixpoint solution of the dataflow problem $\langle C, \mathcal{L}_P\rangle$ and check if the solution is precise enough to answer all our queries. If not, we perform the refinement step to either ascertain a real error in the program, or augment the set of predicates P with new predicates. The process is iterated with this new set of predicates.

EXAMPLE 5: Consider the program of Figure 1. The least dataflow solution obtained using the predicated lattice corresponding to the trivial set of predicates $\{true\}$ is identical to the least solution obtained using the typestate lattice, shown in the second column in Figure 2. At location 8 this solution maps the variable `out` to $\top$ and hence the assertion cannot be proved.

At 4 the dataflow solution maps `out` to `C`, as the file is closed at 3. The lattice value `C` flows from this location along the "else" branch at location 4; at location 6 it becomes $\top$ due to the merge which then flows along the "then" branch at 7 causing the imprecision. The reason for this imprecision is that the dataflow considers the infeasible path corresponding to the "else" branch at location 4 and the "then" branch at location 7. The join at 6 is not the cause as even without the join, we would have the imprecise typestate `C` at the

$$
\begin{array}{lll}
\mathbb{A} & = \{\texttt{null}, \texttt{errp}\} \cup \mathbb{N} & (Address) \\
\mathbb{P} & = \{\mathsf{must}(\mathbb{A}), \mathsf{may}(\mathbb{U}, 2^{\mathbb{A}})\} & (Pointer) \\
\mathbb{U} & = \mathbb{Z} \cup \mathbb{P} \cup \{\top\} & (Value) \\
\Theta & = (\mathbb{N} \rightarrow \mathbb{U}) \cup \{\bot\} & (Lattice\ element)
\end{array}
$$

Figure 3: Types

assertion. Using techniques described in [19] we infer the predicates $\{flag = 0, flag \neq 0\}$, which in the next iteration gives the least solution shown in the third column in Figure 2, which is precise enough to prove the assertion. □

4. SYMBOLIC EXECUTION LATTICE

We now instantiate the generic predicated lattice framework from the previous section by describing a particular lattice for performing dataflow analysis of programs for a wide range of typestate properties.

Statement syntax. We precisely define the set of operations of the imperative language. For clarity, we describe only a subset of the full C grammar here (our implementation supports the full C syntax). The grammar for expressions is defined as follows:

$$
\begin{array}{lll}
\text{(Expressions)} & e & ::= e \oplus e \mid l \mid i \\
\text{(Lvalues)} & l & ::= *l \mid v
\end{array}
$$

for variables or heap cells v, integer constants i, and arithmetic operations $\oplus$.

There are four types of operations: assignments, allocation, deallocation, and assume predicates. An assignment $l := e$ assigns an expression e to the lvalue l. An allocation $l := \texttt{new}\ e$ allocates memory initialized with the value e, and assigns the lvalue l to point to the new memory. A deallocation operation $\texttt{del}\ l$ deallocates memory pointed to by the lvalue l. An assume predicate is either an arithmetic comparison of two arithmetic expressions ($\texttt{assume}(e_1 \sim e_2)$, where $\sim \in \{=, \neq, \leq, \geq, <, >\}$), or pointer equality between pointer-valued expressions ($\texttt{assume}(e_1 = e_2)$ or $\texttt{assume}(e_1 \neq e_2)$). The concrete semantics of the language is given in the standard way, using a store of variables and a heap. We assume that the program halts if a null pointer is dereferenced or if deallocated memory is accessed. We assume in the following that our programs are type-safe, that is, integers and pointers are not mixed in expressions. This allows us to remove error conditions from the operational semantics.

4.1 Lattice Elements

Lattice of values. A lattice element will represent an abstraction of the store and the heap. Figure 3 shows the entities used in defining a lattice element.

The memory (store and heap) is abstracted as a set of (logical) *region names* $\mathbb{N}$. Each region name represents the contiguous area of memory associated with a program variable or a heap pointer dereference. The elements of $\mathbb{N}$ are the program variable names $\mathbb{V}$ and a countable set of heap addresses $\mathbb{H}$ that stand for dynamically allocated memory. A heap region in $\mathbb{H}$ is created whenever new memory is allocated and destroyed whenever memory is deallocated.

An *address* is used to refer to a region name. An address is either a region name from $\mathbb{N}$, or a special symbol $\texttt{null}$ (the null address) that does not point to a valid region, or a special symbol $\texttt{errp}$ that refers to an erroneous address (e.g., an address that has been freed).

Pointer p	$\mathsf{addr}(p)$	$\mathsf{deref}(\theta, p)$
$\mathsf{must}(\rho)$	$\{\rho\}$	$\theta(\rho)$
$\mathsf{may}(v, A)$	A	$\texttt{let}\ A' = \mathsf{addr}(\{\theta(\rho) \mid \rho \in A\})\ \texttt{in}$ v if $v \neq \top$ $\mathsf{may}(A')$ if $v = \top \wedge A' \neq \emptyset$ $\top$ if $v = \top \wedge A' = \emptyset$

Table 1: Semantics for pointer dereferencing

Addresses are referred through pointers. The symbolic execution lattice supports two types of pointer values:

- *Must Pointers* represent pointers which have a single, specific target address. A must pointer corresponds directly to a pointer in the concrete program state and is parameterized by an address. We write $\mathsf{must}(\rho)$ to represent a definite pointer to the address ρ.

- *May Pointers* represent pointers whose target is not precisely known because of imprecision in the analysis. A may-pointer is parameterized by a value and a set of addresses; for a value u and a set of addresses $s \subseteq \mathbb{A}$, $\mathsf{may}(u, s)$ denotes a pointer that may point to any one address in s, but when dereferenced, has the value u. When assigning through a may-pointer, the values at each target address are invalidated and the assigned value stored in the may-pointer itself. This permits a precise value to be returned for future dereferences of the pointer.[1]

The set of pointers is denoted $\mathbb{P}$. A *value* is either an integer, a pointer, or $\top$ (denoting unknown value). We define some helper functions on pointers that will be useful in defining the abstract semantics. The function $\mathsf{addr} : \mathbb{U} \rightarrow 2^{\mathbb{A}}$ returns, for a pointer p, the set of addresses p can point to, and returns the empty set for a non-pointer value. It is naturally extended to sets of values by union: $\mathsf{addr}(U) = \cup_{u \in U} \mathsf{addr}(u)$. Table 1 shows the definitions of addr and deref (described below) based on case analysis on the type of p.

We define the operators $\sqsubseteq$ and $\sqcup$ on the lattice data values $u_1, u_2 \in \mathbb{U}$. The $\sqsubseteq$ operator induces a partial ordering on data values. Any integer value is less than $\top$, and two distinct integers are incomparable. Pointers are ordered based on their sets of potential targets. A pointer p_1 is less than or equal to pointer p_2 if the target address set for p_1 is a subset of the target address set for p_2 (or the target addresses are the same but the value at p_1 is less than or equal to the value at p_2). Any other pair of unequal values are incomparable. Formally, $u_1 \sqsubseteq u_2$ iff (1) $u_1 = u_2$, or (2) $u_2 = \top$, or (3) $u_1 = \mathsf{must}(\rho)$ and $u_2 = \mathsf{may}(v, \{\rho\} \cup A)$, for any set A of addresses, or (4) $u_1 = \mathsf{may}(v_1, A)$, $u_2 = \mathsf{may}(v_2, A \cup A')$, and $v_1 \sqsubseteq v_2$, for sets A and A' of addresses.

The $\sqcup$ operator returns the least upper bound of two val-

[1]Note that we do not model must-pointers with may-pointers whose target is a singleton. This is because singleton may-pointers can be created when modeling arrays and heap regions, which are represented using summary regions (see Section 4.3).

ues based on this partial ordering. Formally, for $u_1, u_2 \in \mathbb{U}$:

$$u_1 \sqcup u_2 \equiv \begin{cases} u_1 & \text{if } u_1 = u_2 \\ \top & \text{if } (u_1 = \top \vee u_2 = \top) \\ \mathsf{may}(u', A_1 \cup A_2) & \text{if } u_1 = \mathsf{may}(u', A_1) \wedge \\ & \quad u_2 = \mathsf{may}(u', A_2) \\ \mathsf{may}(\top, A_1 \cup A_2) & \text{if } u_1, u_2 \in \mathbb{P} \wedge u_1 \neq u_2 \wedge \\ & \quad A_1 = \mathsf{addr}(u_1) \wedge \\ & \quad A_2 = \mathsf{addr}(u_2) \\ \top & \text{otherwise} \end{cases}$$

Abstract states. An abstract state θ is either (1) a partial mapping $\mathbb{N} \to \mathbb{U}$ from the set of region names $\mathbb{N}$ to the set of values $\mathbb{U}$, or (2) the element $\perp$. The set of abstract states, denoted Θ is the set of all lattice elements. For an abstract state θ, let $dom(\theta)$ denote the set of names in the domain of θ. For $\rho \in dom(\theta)$, we write $\theta(\rho)$ for the image of ρ under θ. The function $\mathsf{deref} : \Theta \times \mathbb{P} \to \mathbb{U}$, defined in Table 1 returns a value pointed to by a pointer p. For an abstract state θ, a region ρ, and value u, we write $\theta[\rho \mapsto u]$ for the abstract state that maps ρ to u, and agrees with θ on all other regions. We write $\theta \cup \{(\rho, u)\}$ for the abstract state with domain $dom(\theta) \uplus \{\rho\}$ (i.e., ρ is a fresh name), that maps ρ to u, and agrees with θ on $dom(\theta)$. We write $\theta \setminus \{\rho\}$ for the restriction of θ to $dom(\theta) \setminus \{\rho\}$.

Each abstract state θ represents a set of concrete data states $[\![\theta]\!]$ defined as:

$$[\![\theta]\!] \equiv \Big\{ s \in S \mid \forall \rho \in dom(\theta). \Big(s(\rho) \in val(\theta(\rho)) \wedge \\ (\theta(\rho) = \mathsf{may}(u, A) \Rightarrow \exists a \in A. s(a) \sqsubseteq u) \Big) \Big\}$$

where

$$val(u) \equiv \begin{cases} \{u\} & \text{if } u \in \mathbb{Z} \\ \mathsf{addr}(u) & \text{if } u \in \mathbb{P} \\ \mathbb{U} & \text{if } u = \top \end{cases}$$

A lattice element restricts the set of concrete data states to those where, for each region ρ defined by the mapping θ, the associated value is one of the possible values represented by $\theta(\rho)$. In addition, if the value is a may-pointer $\mathsf{may}(u, A)$, a target of the pointer (from the set A) must contain the value u. The lattice element $\perp$ represents the empty set: $[\![\perp]\!] = \emptyset$. The element $\top$ is the function that maps all region names to $\top$.

Ordering and join. The partial ordering $\sqsubseteq$ on values is extended to lattice elements by defining $\theta_1 \sqsubseteq \theta_2$ iff for all names $\rho \in \mathbb{N}$, we have $\theta_1(\rho) \sqsubseteq \theta_2(\rho)$ (where, for $i = 1, 2$, we assume $\theta_i(\rho) = \top$ for any $\rho \notin dom(\theta_i)$). Notice that this definition ensures that $\theta_1 \sqsubseteq \theta_2$ iff $[\![\theta_1]\!] \subseteq [\![\theta_2]\!]$. The join $\theta_1 \sqcup \theta_2$ of lattice elements θ_1 and θ_2 is defined as the function $\lambda\rho.\theta_1(\rho) \sqcup \theta_2(\rho)$, where (for $i = 1, 2$) we assume $\theta_i(\rho) = \top$ for all $\rho \notin dom(\theta_i)$.

4.2 Abstract Semantics

The operation $\widehat{\mathsf{SP}}(\theta, l)$ returns a lattice element θ' which is an over-approximation of $\mathsf{SP}([\![\theta]\!], l)$. The semantics of the post operation are defined using evaluation rules below.

Expression evaluation. Table 2 lists the semantics for evaluating an expression e in an environment θ. The evaluation rules are given as a big step semantics with the evaluation operator $\mathsf{eval} : \mathsf{Exp} \times \Theta \to \mathbb{U}$, that takes an expression and a lattice element and produces a value. We write $e \rightsquigarrow_\theta u$ to denote $\mathsf{eval}(e, \theta) = u$, and we omit the subscript θ when it is clear from the context. The rules are standard. Additionally, we define the operator $\mathsf{addrOf} : \mathsf{Exp} \times \Theta \to \mathbb{A}$, that takes an lvalue and an abstract state and returns its address. The evaluation rules for addrOf are similar to eval, and are omitted.

Assignments and allocation. Table 3 defines the abstract post operation for statements. Since assignment statements may change the heap, the abstract lattice must consistently approximate the concrete store through assignments. We use two helper functions $\mathsf{invalidate} : \Theta \times 2^{\mathbb{A}} \times \mathbb{U} \to \Theta$ and $\mathsf{ptsTo} : \mathbb{A} \to 2^{\mathbb{P}}$ to do this. The function $\mathsf{invalidate}(\theta, s, u)$ returns $\theta[\wedge_{\rho \in \mathsf{addr}(s)} \rho \mapsto \mathsf{eval}(\theta, \rho) \sqcup u]$. The function $\mathsf{ptsTo}(\rho)$ returns the set of all pointers that may be pointing to the address ρ.

Rule 1 defines the semantics of assignments where the lvalue is a variable. Such an assignment adds a new mapping between the variable and the evaluated value of the target. Rules 2 and 3 define the semantics of assignments where the target expression is a pointer dereference. Rules 4 and and 5 define the semantics of allocation and deallocation of memory.

Rule 3 is of particular interest as it provides strong update semantics when updating through an imprecise pointer. When the target expression evaluates to a may-pointer whose address set is s, a mapping is added between addrOf applied to the assignment's target expression and a may-pointer which points to s, but has the assignment's source expression as its target value. For soundness, the values at all addresses in s are invalidated (integers set to $\top$ and pointers converted to may-pointers, adding any pointers referenced by e_2). Using may-pointers provides a mechanism for the dataflow analysis to infer some local non-aliasing, similar to *restrict* [2], without programmer annotations.

Assumes. Rules 6-7 in Table 3 define the semantics for assume predicates. If the predicate evaluates to false (rule 6), the resulting lattice element is $\perp$, indicating an inconsistency. This indicates that the path is infeasible. If an inequality does not evaluate to 0, the store remains unchanged (rule 7).

In addition, we implement strengthening of the lattice after certain assume statements. If an equality $e_1 = e_2$ does not evaluate to 0, the values of the two expressions are inspected. If e_1 evaluates to the address ρ, and e_2 evaluates to a value $u \sqsubseteq \theta(\rho)$ in the value ordering, a new mapping is created between the region ρ and the value u, that is, the new lattice is $\theta[\rho \mapsto u]$. The less-than comparison is used to ensure that the mapping for ρ is changed only when the new value is more precise. Otherwise, the store is left unchanged. For example, if the current lattice maps variable x to $\top$, and encounters the assume $\mathtt{assume}(x = 0)$, then after the assume, the new lattice maps x to 0.

PROPOSITION 3. *For any operation* $\mathsf{op} \in Ops$ *and any lattice element* θ, *we have* $[\![\widehat{\mathsf{SP}}(\theta, \mathsf{op})]\!] \supseteq \mathsf{SP}([\![\theta]\!], \mathsf{op})$.

THEOREM 1. *The set of abstract states, together with the constants* $\top$, $\perp$, *functions* $\sqcup$, $\widehat{\mathsf{SP}}$, $[\![\cdot]\!]$, *and relation* $\sqsubseteq$ *forms a semi-lattice.*

Rule	Expression e	Condition	$\mathsf{eval}(\theta, e)$
1	v		$\theta[v]$
2	$*l$	$l \rightsquigarrow p : \mathbb{P}$	$\mathsf{deref}(\theta, p)$
3	$i \in \mathbb{Z}$		i
4	$e_1 \oplus e_2$	$e_1 \rightsquigarrow i_1 : \mathbb{Z} \wedge e_2 \rightsquigarrow i_2 : \mathbb{Z}$	$i_1 \oplus i_2$
5	$e_1 \oplus e_2$	$(e_1 \rightsquigarrow \top \vee e_2 \rightsquigarrow \top)$	$\top$
6	$e_1 \sim e_2$	$e_1 \rightsquigarrow i_1 : \mathbb{Z} \wedge e_2 \rightsquigarrow i_2 : \mathbb{Z} \wedge i_1 \sim i_2$	1
7	$e_1 \sim e_2$	$e_1 \rightsquigarrow i_1 : \mathbb{Z} \wedge e_2 \rightsquigarrow i_2 : \mathbb{Z} \wedge i_1 \not\sim i_2$	0
8	$e_1 \sim e_2$	$(e_1 \rightsquigarrow \top \vee e_2 \rightsquigarrow \top)$	$\top$
9	$e_1 = e_2$	$e_1 \rightsquigarrow \mathsf{must}(a_1) \wedge e_2 \rightsquigarrow \mathsf{must}(a_2) \wedge a_1 = a_2$	1
10	$e_1 = e_2$	$e_1 \rightsquigarrow p_1 : \mathbb{P} \wedge e_2 \rightsquigarrow p_2 : \mathbb{P} \wedge (\mathsf{addr}(p_1) \cap \mathsf{addr}(p_2) \neq \emptyset) \wedge$ $(p_1 \neq \mathsf{must}(a_1) \vee p_2 \neq \mathsf{must}(a_2))$	$\top$
11	$e_1 = e_2$	$e_1 \rightsquigarrow p_1 : \mathbb{P} \wedge e_2 \rightsquigarrow p_2 : \mathbb{P} \wedge (\mathsf{addr}(p_1) \cap \mathsf{addr}(p_2) = \emptyset$	0

Table 2: Expression evaluation

Rule	op	Conditions	$\widehat{\mathsf{SP}}(\theta, \mathtt{op})$
1	$v := e$	$e \rightsquigarrow u : \mathbb{U}$	$\theta[v \mapsto u]$
2	$*e_1 := e_2$	$e_1 \rightsquigarrow \mathsf{must}(\rho) \wedge \rho \neq \mathtt{null} \wedge \rho \neq \mathtt{errp} \wedge e_2 \rightsquigarrow u : \mathbb{U}$	$\theta[\rho \mapsto u]$
3	$*e_1 := e_2$	$e_1 \rightsquigarrow \mathsf{may}(u', s) \wedge e_2 \rightsquigarrow u$	$\mathsf{invalidate}(\theta, s, u)[\mathsf{addrOf}(*e_1) \mapsto \mathsf{may}(u, s)]$
4	$l := \mathtt{new}\ e$	$e \rightsquigarrow u$	$(\theta \cup \{(\rho, u)\})[\mathsf{addrOf}(l) \mapsto \mathsf{must}(\rho)]$, ρ fresh
5	$\mathtt{del}\ l$		$\widehat{\mathsf{SP}}(\theta, l := \mathtt{errp})$
6	$\mathtt{assume}(e_1 \sim e_2)$	$(e_1 \sim e_2) \rightsquigarrow 0$	$\perp$
7	$\mathtt{assume}(e_1 \sim e_2)$	$(e_1 \sim e_2) \rightsquigarrow 1 \vee (e_1 \sim e_2) \rightsquigarrow \top$	θ

Table 3: Abstract semantics $\widehat{\mathsf{SP}}(\theta, \mathtt{op})$

4.3 Dynamic Allocation

Our definition so far provides a lattice that can represent potentially infinite heaps, since we allow dynamic allocation of memory. We cannot bound the size of the domain of region names if we have memory allocation (see rule 5 in Table 3). In order to write a dataflow analysis based on this lattice, we must abstract the heap to a finite structure. We take the standard approach of summarizing heap cells based on an equivalence relation on dynamically allocated regions.

Formally, we extend the set of addresses with *summary addresses* $\mathbb{S}$,

$$\mathbb{A} = \{\mathtt{null}, \mathtt{errp}\} \cup \mathbb{N} \cup \mathbb{S}.$$

Summary addresses denote equivalence classes of dynamically allocated heap names. Formally, let E be a partition on regions with a finite index. Intuitively, summary addresses are E-equivalence classes denoting sets of dynamically allocated regions. If the E-equivalence class $\hat{\rho}$ has exactly one region, then $\hat{\rho}$ behaves like a regular address, and we allow must-pointers to it, and strong updates on it. If the E-equivalence class $\hat{\rho}$ has more than one possible region, then $\hat{\rho}$ behaves as a set of addresses, and we only allow may-pointers to it, and all updates are weak. We extend the ordering and join operations to take care of summary addresses, using $\mathsf{addr}(\hat{\rho}) = \mathsf{addr}(\hat{\rho}')$ if $\hat{\rho}$ and $\hat{\rho}'$ denote the same E-equivalence class, and $\mathsf{addr}(\hat{\rho}) \cap \mathsf{addr}(\hat{\rho}') = \emptyset$ if they denote different E-equivalence classes.

We alter the abstract semantics of the allocation command $\mathtt{l} := \mathtt{new}\ \mathtt{e}$ in the following way. Let $e \rightsquigarrow_\theta u$, and let $\hat{\rho}$ be the E-equivalence class of u. Then $\widehat{\mathsf{SP}}(\theta, \mathtt{l} := \mathtt{new}\ \mathtt{e})$ is the lattice element

$$\theta'[\mathsf{addr}(l) \mapsto \mathsf{summary}(u, \hat{\rho})]$$

where θ' equals $\theta[\hat{\rho} \mapsto \theta(\hat{\rho}) \sqcup u]$ if $\hat{\rho} \in dom(\theta)$ and θ' equals $\theta \cup \{(\hat{\rho}, u)\}$ otherwise; and $\mathsf{summary}(u, \hat{\rho})$ returns $\mathsf{may}(u, \{\hat{\rho}\})$ if the E-equivalence class $\hat{\rho}$ has more than one possible region, and returns $\mathsf{must}(\hat{\rho})$ if $\hat{\rho}$ represents exactly one region. We now give two examples of partitions E.

1. **Merge.** Consider the partition of dynamically allocated regions that merges all dynamically allocated regions at the same CFA edge (i.e, any two regions allocated at the same site are equivalent, while any two regions allocated at different sites are not equivalent). This is the standard abstraction in program analysis (see, e.g., [17]).

2. **Merge modulo predicates.** Let P be a set of predicates, and consider the set of functions from P to the set $\{0, 1, \top\}$ that map each predicate to a truth value 0 (false), 1 (true), or $\top$ (unknown). Consider the partition that merges all dynamically allocated regions at the same CFA edge that agree on the predicate mapping. This is the 3-valued abstraction in shape analysis [27].

Example 6: Consider the program `Example2` in Figure 4(a). The program allocates memory dynamically in a variable `x` inside a loop, initialized with 0, sets $*$`x` to 1, and asserts that $*$`x` is 1. It then sets $*$`x` to 0. Figure 4(b) shows the least fixpoint solution of the dataflow problem for the CFA of `Example2` with the symbolic execution lattice. Since the dataflow fact at location 4 maps `x` to $\mathsf{may}(1, \{\hat{\rho}\})$, we have that $*$`x` $= 1$ at this location. Thus the assertion holds. Notice that the may-pointer preserves the information that $*$`x` is 1, even though the abstract location $\hat{\rho}$ is mapped to

```
Example2() {
1:  while (...)  {
2:       x := new 0;
3:       *x := 1;
4:       assert(*x=1);
5:       *x := 0;
    }
}
```

1	$\mathtt{x} \mapsto \mathtt{errp}$
2, 3	$\mathtt{x} \mapsto \mathsf{may}(\top, \{\hat{\rho} : \mathbb{P}\}), \hat{\rho} \mapsto \top$
4, 5	$\mathtt{x} \mapsto \mathsf{may}(1, \{\hat{\rho} : \mathbb{P}\}), \hat{\rho} \mapsto \top$

Figure 4: (a) A program and (b) the least fixpoint dataflow solution

$\top$ (since $\hat{\rho}$ represents multiple dynamically allocated locations, some of which contain value 0, and others value 1). In contrast, a dataflow analysis that merges all dynamically allocated memory locations, and tracks values of abstract locations (*e.g.,* [17]) cannot prove the assertion on line 4, since the abstract location $\hat{\rho}$ is mapped to $\top$, and we only know that x may point to $\hat{\rho}$. □

5. EXPERIMENTS

We have implemented the predicated lattice framework with the symbolic execution lattice in the Blast software verification tool [20].

Implementation Details. The formalization of the symbolic execution lattice presented in Section 4 did not include arrays and structures, key components of most non-trivial C programs. The implementation models both arrays and structures.

Array values are represented by a single summary node computed by taking the join of all array elements. In C, a program variable may be explicitly declared as an array or any contiguous area of memory can be treated as an array through the use of pointer arithmetic. As a result, a lattice element must track whether each region is an array (type information is not sufficient). We extend the set of addresses to include offsets from a base address. Pointer arithmetic is modeled conservatively, so after a pointer arithmetic operation, we assume that the resulting pointer can point to any part of the array.

To represent structures, we extend the set of lattice elements by allowing nesting. A nested lattice element represents a C structure. The names and values of a nested lattice element represent the structure's field names and field values respectively. C structures are modeled precisely by the lattice: each member of a structure is treated as an independent subregion. The array property of a region is inherited by its members, if a structure is stored in an array region, all of its members are treated as arrays as well.

Since the lattice tracks alias information, we use it to answer aliasing queries during the analysis. For example, this is useful in disambiguating function-pointer calls at analysis time: for a function pointer whose value is $\mathsf{may}(F)$, we only call the functions in the set F.

In our implementation, we modify the abstraction of dynamically allocated memory locations from the merge strategy as follows. We partition dynamically allocated regions according to allocation site. Within each partition, we again partition regions into two sets: the latest allocated region, and the set of all previously allocated regions. This is not a (time invariant) partition of regions as defined above. However, at each point of the execution, this defines a partition of the heap. With this strategy, the abstract post computation is more involved, since at each allocation, the last allocated region must be merged with all the regions allocated at that site before it, and some must pointers (pointing to the last allocated region) have to be modified to may pointers after the merge. We omit the implementation details.

Kernel API in Windows drivers. We checked a set of Microsoft Windows device drivers for a typestate property related to the proper handling of I/O requests. The property is a finite-state automaton with 22 states [4]. The source code for these drivers ranged from 14,000 to 138,000 lines of (preprocessed) C code. These experiments were run on a Dell PowerEdge 1800 with two 3.6Ghz Xeon processors and 5 GB of memory. We summarize the results in Table 4. The first two columns give the name of the driver and the number of CFA nodes in the reachable call graph. There are three sets of numbers: results of running a pure predicate abstraction based algorithm implemented in the Blast model checker [20] without the symbolic execution lattice from Section 4, running the predicated dataflow algorithm with the symbolic execution lattice, and running the counterexample-guided abstraction refinement algorithm with the lattice when the specification predicates are provided to the algorithm (ESP mode). There are four numbers for each of the first two sets: the number of iterations ('Iter'), the number of refinements ('Ref'), the total number of predicates added ('Pred'), and the time taken in seconds. The final numbers ('Spec') shows the number of refinements and additional predicates added when we start with all the specification predicates.

The number of iterations is the number of abstract post computations that were computed. The number of refinements can be more than the number of predicates, since we use a localized predicate discovery algorithm [19], and the same predicates may be found at different program locations in different refinement steps.

The algorithm with the lattice outperforms the older version of Blast in all cases. The number of refinement steps and correspondingly, the number of predicates, are significantly lower, since the symbolic execution lattice can rule out many infeasible paths directly. This translates to faster running times.

Finally, we ran the predicated dataflow algorithm, giving the set of predicates from the specification as the initial set of predicates. These predicates constitute the state of the specification automaton. The ESP algorithm [13] would run a predicated dataflow analysis over this fixed set of predicates; any additional refinement required would show up as a false positive in their analysis. For this more complicated safety property, we show that the predicated lattice with the specification predicates is not precise enough: for all the examples, the algorithm has to perform additional refinement steps. This shows that ESP would flag false positives in all these examples. In the 'Spec' column, we provide the additional number of refinement steps performed in each case (column 'Ref') and the additional number of predicates found beyond the specification predicates (column 'Preds'). The number of refinements when the specification predicates were given is different from the case when no predicates are initially supplied. This is because different sets of infeasible paths are considered for refinement in the two cases.

Acknowledgments. This research was supported in part by the grant NSF CCR-0427202.

Program	CFA nodes	No lattice				Lattice				Spec	
		Iter	Ref	Preds	Time (s)	Iter	Ref	Preds	Time (s)	Ref	Preds
`diskperf`	549	34035	164	154	831	4860	33	31	21	47	47
`cdaudio`	968	22964	68	95	517	6080	27	25	18	145	108
`floppy`	1039	21859	125	123	422	5624	42	42	35	54	60
`parclass`	1663	71480	177	185	3325	8630	38	42	89	103	116
`parport`	2518	113420	184	212	5829	43135	37	49	314	146	163

Table 4: Experimental Results for Windows drivers

6. REFERENCES

[1] T. Agerwala and J. Misra. Assertion graphs for verifying and synthesizing programs. Technical Report 83, University of Texas, Austin, 1978.

[2] A. Aiken, J.S. Foster, J. Kodumal, and T. Terauchi. Checking and inferring local non-aliasing. In *PLDI 03: Programming Language Design and Implementation*, pages 129–140. ACM, 2003.

[3] R. Alur, A. Itai, R.P. Kurshan, and M. Yannakakis. Timing verification by successive approximation. *Information and Computation*, 118(1):142–157, 1995.

[4] T. Ball and S.K. Rajamani. Personal communication.

[5] T. Ball and S.K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL 02: Principles of Programming Languages*, pages 1–3. ACM, 2002.

[6] R. Bodik, R. Gupta, and M.L. Soffa. Refining dataflow information using infeasible paths. In *FSE 97: Foundations of Software Engineering*, LNCS 1301, pages 361–377. Springer, 1997.

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

[8] S. Chaki, E.M. Clarke, A. Groce, and O. Strichman. Predicate abstraction with minimum predicates. In *CHARME 03: Correct Hardware Design and Verification*, LNCS 2860, pages 19–34. Springer, 2003.

[9] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV 00: Computer-Aided Verification*, LNCS 1855, pages 154–169. Springer, 2000.

[10] E.M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS 04: Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 2988, pages 168–176. Springer, 2004.

[11] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In *POPL 77: Principles of Programming Languages*, pages 238–252. ACM, 1977.

[12] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL 79: Principles of Programming Languages*, pages 269–282. ACM, 1979.

[13] M. Das, S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. In *PLDI 02: Programming Language Design and Implementation*, pages 57–68. ACM, 2002.

[14] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[15] M. Dwyer and L. Clarke. A flexible architecture for building dataflow analyzers. In *ICSE 96: International Conference on Software Engineering*, pages 554–564. ACM, 1996.

[16] M.B. Dwyer, L.A. Clarke, J.M. Cobleigh, and G. Naumovich. Flow analysis for verifying properties of concurrent software systems. *ACM Transactions on Software Engineering Methodology*, 13(4):359–430, 2004.

[17] J.S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *PLDI 02: Programming Language Design and Implementation*, pages 1–12. ACM, 2002.

[18] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV 97: Computer-aided Verification*, LNCS 1254, pages 72–83. Springer, 1997.

[19] T.A. Henzinger, R. Jhala, R. Majumdar, and K.L. McMillan. Abstractions from proofs. In *POPL 04: Principles of Programming Languages*, pages 232–244. ACM, 2004.

[20] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL 02: Principles of Programming Languages*, pages 58–70. ACM Press, 2002.

[21] L.H. Holley and B.K. Rosen. Qualified data flow problems. *IEEE Transactions on Software Engineering*, SE-7:60–78, 1981.

[22] Gary Kildall. A unified approach to global program optimization. In *POPL 73: Principles of Programming Languages*, pages 194–206. ACM, 1973.

[23] B. Murphy. *Frameworks for precise program analysis*. PhD thesis, Stanford University, 2001.

[24] M. Musuvathi, D.Y.W. Park, A. Chou, D.R. Engler, and D.L. Dill. CMC: A pragmatic approach to model checking real code. In *OSDI 02: Operating Systems Design and Implementation*. ACM, 2002.

[25] F. Nielson. Expected forms of data flow analyses. In *Programs as Data Objects*, LNCS 217, pages 172–191. Springer, 1985.

[26] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL 95: Principles of Programming Languages*, pages 49–61. ACM, 1995.

[27] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL 99: Principles of Programming Languages*, pages 105–118. ACM, 1999.

[28] R.E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986.

[29] P. Tu and D. Padua. Automatic array privatization. In *Compiler Optimizations for Scalable Parallel Systems Languages*, pages 247–284, 2001.

[30] M.N. Wegman and K. Zadek. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13:181–210, 1991.

[31] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *POPL 05: Principles of Programming Languages*, pages 351–363. ACM, 2005.