Probabilistic, Modular and Scalable Inference of Typestate Specifications

Nels E. Beckman
Carnegie Mellon University
nbeckman@cs.cmu.edu

Aditya V. Nori
Microsoft Research India
adityan@microsoft.com

Abstract

Static analysis tools aim to find bugs in software that correspond to violations of specifications. Unfortunately, for large and complex software, these specifications are usually either unavailable or sophisticated, and hard to write.

This paper presents ANEK, a tool and accompanying methodology for inferring specifications useful for modular typestate checking of programs. In particular, these specifications consist of pre and postconditions along with aliasing annotations known as access permissions. A novel feature of ANEK is that it can generate program specifications even when the code under analysis gives rise to conflicting constraints, a situation that typically occurs when there are bugs. The design of ANEK also makes it easy to add heuristic constraints that encode intuitions gleaned from several years of experience writing such specifications, and this allows it to infer specifications that are better in a subjective sense. The ANEK algorithm is based on a modular analysis that makes it fast and scalable, while producing reliable specifications. All of these features are enabled by its underlying probabilistic analysis that produces specifications that are very *likely*.

Our implementation of ANEK infers access permissions specifications used by the PLURAL [5] modular typestate checker for Java programs. We have run ANEK on a number of Java benchmark programs, including one large open-source program (approximately 38K lines of code), to infer specifications that were then checked using PLURAL. The results for the large benchmark show that ANEK can quickly infer specifications that are both accurate and qualitatively similar to those written by hand, and at 5% of the time taken to manually discover and hand-code the specifications.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Statistical methods

General Terms Algorithms, Verification

Keywords aliasing, inference, object protocol, ownership, permission, typestate, specification

1. Introduction

Developing correct and reliable software is a hard problem, one that is supported by various analysis tools for improving software quality and reliability. Even though these tools are largely automated, many require user-provided specifications that describe the property or protocol that the code under analysis is required to satisfy [2, 5, 8, 11] and this greatly limits their application in practice. However, writing good specifications is a hard and laborious task that also requires some degree of expertise on the codebase under analysis.

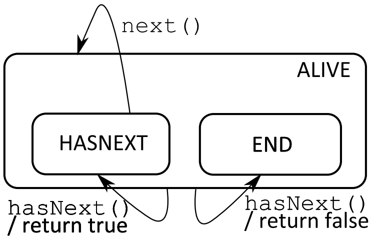

Figure 1. The iterator protocol.

In this paper we present ANEK[1], a tool and an accompanying methodology for inferring specifications necessary for modular typestate [18] checking. Given an API annotated with typestate specifications by its developers, ANEK allows users of that API to quickly infer the specifications needed in their code in order to modularly check API conformance.

Specifically, ANEK infers access permission specifications [5], specifications which, in addition to encoding abstract states of objects, also encode aliasing information, which allows for sound modular checking. Our inference algorithm is novel because it combines both the logical facts relating to permission creation and destruction as well as heuristics describing likely specifications, and does so using a scheme of *probabilistic* constraints. The result is an inference that is faster, generates better specifications and scales better than more traditional approaches.

To illustrate our point, consider an example. A programmer wants to ensure correct use of Java's iterator API, modeled in Figure 1 and annotated by its developers with access permission specifications in Figure 2. Its specification says that in order to call the `next` method, the receiver object must be known to be in the "HASNEXT" abstract state, which a client can be sure the object is in if the `hasNext` method returns true. The `Collection` interface is also specified. It says that when the `iterator` method is called, a unique (unaliased) iterator will be returned in the "ALIVE" state.[2]

Our hypothetical programmer wants to ensure that her simple spreadsheet application, partially given in Figure 3, is always using

PLDI'11, June 4–8, 2011, San Jose, California, USA.
Copyright © 2011 ACM 978-1-4503-0663-8/11/06…$10.00

[1] *Anek* is the Hindi word meaning *plural*.

[2] The "ALIVE" state in the PLURAL methodology is the root of the state hierarchy, so this statement is equivalent to saying the iterator is not in any state of interest.

```
interface Iterator<T> {
  @Spec(requires="full(this) in HASNEXT",
        ensures="full(this) in ALIVE")
  T next();

  @Spec(requires="pure(this) in ALIVE",
        ensures="pure(this)")
  @TrueIndicates("HASNEXT")
  @FalseIndicates("END")
  boolean hasNext();
}

interface Collection<T> extends Iterable<T> {
  @Spec(ensures="unique(result) in ALIVE")
  Iterator<T> iterator();
  // Continues...
}
```

Figure 2. Iterator and collection interfaces annotated with access permission specifications.

```
 1 class Row {
 2   Collection<Integer> entries;
 3
 4   Iterator<Integer> createColIter() {
 5     return entries.iterator();
 6   }
 7   void add(int val){...}
 8   // Continues...
 9 }
10
11 // Many similar uses of iterator exist
12 Row copy(Row original) {
13   Iterator<Integer> iter =
14     original.createColIter();
15   Row result = new Row();
16   while( iter.hasNext() ) {
17     result.add(iter.next());
18   }
19 }
20
21 @Test
22 void testParseCSV() {
23   Row r1 = parseCSVRow("1,2,3,4");
24   Row r2 = parseCSVRow("4,6,7,8");
25   int sum = r1.createColIter().next() +
26             r2.createColIter().next();
27   assert(sum,5);
28 }
```

Figure 3. An application using the iterator API for which specifications are needed.

the iterator API correctly, and wants to use the PLURAL modular typestate checker [5] to do so, but does not want to spend time writing specifications. In this program, a Row class acts as an abstraction of a spreadsheet row, and wraps an instance of the Collection class. Notably, the createColIter method on line 4 simply returns the result of the collection's iterator method. A number of uses of the column iterator in this program are similar to its use in the copy method, line 12, where the iterator's next method is called in a loop, and only when calls to the hasNext method return true. However, one use of the column iterator occurs in a test method, testParseCSV on line 22, which tests the ability to input data in CSV format. In this use, the next method is immediately called on iterator returned by the createColIter method (line 25). The reasoning is that, assuming the program is implemented correctly, the parseCSVRow method will return a non-empty row for the given inputs.

Here, let us concentrate on the specification to be inferred for the return value of the createColIter method (line 4). Traditional approaches would have difficulty inferring specifications here because different uses of the createColIter method give rise to conflicting constraints. The use of the method on line 25 implies that the returned value must be in the "HASNEXT" state. At the same time, the existing specification of the iterator method indicates that the iterator is returned in the "ALIVE" state, and this is consistent with its use in copy and all of the similar uses not shown. A traditional analysis would generate two constraints containing conflicting information, satisfaction of these constraints with a Boolean constraint solver would be impossible, and no specification would be produced.

In contrast, our approach builds logical constraints on top of probabilities, so that conflicting facts can coexist. ANEK will generate a constraint at line 25 saying that the return value of createColIter must be in the "HASNEXT" state *with high probability*, and will generate constraints at line 5, line 16 and all similar sites saying that the return value must be in the "ALIVE" state *with high probability*. The evidence for the "HASNEXT" state is outweighed by the evidence against it, and the "ALIVE" abstract state will be chosen for the return value. Such an inferred specification may cause a typestate checker to issue a warning on line 25.

While it may seem somewhat strange to infer specifications that are only mostly correct, and may indeed lead to analysis warnings, in practice [6] some number of false positives are inevitable when analyzing any large program, and practically-motivated programmers still desire to verify as much of their code as is possible. Note also that the PLURAL static analysis is sound, so there are no safety concerns due to the approximative nature of the inference.

As a further benefit, ANEK's system of probabilistic constraints allows us to easily incorporate heuristic information from a variety of sources, heuristics which can be used to guide the inference process. This issue is relevant for inferring the specification on the returned value of the createColIter method. Each call to the next method of the iterator requires a full aliasing permission, as indicated by the specification in Figure 2. In the access permissions methodology, this indicates an exclusive modifying permission that can coexist with other read-only permissions. However, another permission, unique, which indicates an absence of any other references and is returned by the iterator method of the collection, is stronger than full, and can be used to satisfy the precondition of the next method as well. The question becomes, should the createColIter method be inferred to return a permission of type full or unique, in the absence of any other constraints? ANEK includes the heuristic that all method names beginning with "create" return unique permissions with higher probability, since they are, in practice, often wrappers for constructors. As returned permissions go, unique is the best choice whenever possible because it gives the strongest guarantees to callers. This heuristic applies to the createColIter method, and therefore is used to determine that the return permission specification should be unique.

In addition to encoding heuristics and allowing the generation of specifications in the face of conflicting constraints, probabilistic constraints also enable ANEK to perform inference in a modular fashion. One problem with many existing inference algorithms is that they lack scalability, since the entire program must be analyzed at once. ANEK, in contrast, is a modular algorithm that generates method summaries in the form of probabilities representing likely method specifications, which can then be refined as methods are analyzed and reanalyzed in an iterative fashion. Since the analysis is approximate, it suffices to run the inference algorithm for a fixed number of iterations without reaching a fixpoint (as is usually required by traditional iterative algorithms). Varying the number of iterations allows for a trade-off between specification accuracy and scalability.

ANEK infers specifications required by the PLURAL modular static typestate checker [3, 5]; these are method pre and postconditions that describe both the abstract states parameters must inhabit and aliasing permissions. The probabilistic constraints encode: (a) logical rules which determine how permissions can be used, and (b) heuristic rules which encode the most common or "best" specifications in various scenarios. These constraints form a model that represents a probabilistic view of the space of all possible specifications. A solution to this model results in specifications that are very likely. This paper makes the following contributions:

- We have developed a probabilistic specification inference algorithm that combines logical rules (that encode program invariants) with heuristic rules (that encode program intuitions) for generating typestate specifications that in practice is fast and generates good specifications.
- The algorithm generates probabilistic method summaries which enable a modular analysis that can scale the inference to large programs.
- We have evaluated ANEK on a number of small benchmark programs and one large open-source program containing 38,483 lines of code. The results for this large benchmark show that ANEK can quickly infer specifications that are both accurate and qualitatively similar to those written by hand, and at 5% of the time taken to manually discover and hand-code the specification. As a consequence, these programs can be verified by PLURAL automatically, with no user provided specifications.

This paper is organized as follows. Section 2 gives an overview of the problem and also some background on the PLURAL system. Section 3 defines the abstraction and constraint system in ANEK together with the inference algorithms. Section 4 describes our empirical evaluation of ANEK. And finally, Sections 5 and 6 describe the related work and conclusion respectively.

2. Background and Goals

The purpose of ANEK is to infer typestate specifications, which are then fed to PLURAL [3, 5], a static modular typestate checker for Java programs. With PLURAL, programs can be checked for correct protocol usage by examining one method at a time without ever having to reconsider methods previously analyzed (in a manner analogous to type-checking in most languages). In PLURAL, reference types are supplemented with specifications that act as type refinements. These refinements are flow-sensitive, so that they can change at each step of the program as objects change state (for example, as a file might change from open to closed). These refinements are called *access permissions*. An access permission includes information about the abstract state of an object and a succinct description of which operations any existing aliases to that object are permitted to perform. The need for tracking the abstract state of the reference is mostly straightforward. As a program reference transitions through the body of the method, PLURAL keeps track of the abstract state of the object as methods are called and as fields are read and assigned. When a method is called in the method currently under analysis, PLURAL will check the abstract states of each of the arguments and make sure they match the required states for each of the parameters to the method, as indicated by the method's specification.

But a modular checker also needs to know something about how objects might be aliased if it is to perform a sound static analysis. In the absence of any aliasing information, a sound modular checker would be forced to conservatively assume that just about every method call may modify the abstract states of all of the objects in the static context through other program references. The aliasing summary contained in each access permission informs the analysis when it can be sure an object is not being modified through other aliases, and when it must assume the object is. Because the aliasing summaries are checked for consistency, the entire process is sound.

The aliasing summaries used by PLURAL are called *permission kinds*, and there are five of them. Each permission kind associated with a reference determines whether or not modification can be performed through that reference, and whether or not other aliases, if they exist, can read or modify. The five permission kinds are summarized in Figure 4.

	Other aliases		
This Ref.	N/A	Can Read	Can Write
Can Read	Unique	Immutable	Pure
Can Write	Unique	Full	Share

Figure 4. The five permission kinds.

At the point in code where an object is constructed, the new reference is associated with a unique permission. From that point forward, new aliases can be introduced through a process known as *splitting*. For example, from a reference with unique permission, a new modifying alias can be introduced by destroying the original unique permission and creating two new share permissions, one for the original reference and one for the new reference. Alternatively, a single unique permission could be exchanged for one full permission and multiple pure permissions. PLURAL makes sure that these splits are done soundly, in a manner that respects the meaning of each permission introduced. Additionally, permissions are associated with fractional values which allow multiple weaker permissions to be combined into stronger ones in a process known as *merging*. This feature comes from existing work on fractional permissions [7].

The most common way to use PLURAL is for framework and library designers to annotate their APIs with access permission specifications. API clients can run PLURAL on their code to ensure that they are using the APIs correctly. If API objects are passed as method parameters or stored as fields in a client's program, then she may need to add a specification to her own program in order to inform the analysis what permissions are available.

Figure 2 shows a specification for the Iterator interface using access permissions. The `next` method requires an exclusive modifying permission (full) to the receiver which it returns to the caller. Before the call, the receiver must be in the "HASNEXT" state. The `hasNext` method, which does not need to modify the iterator object, takes a pure permission, which it returns to the caller.

2.1 Goals for ANEK

While the PLURAL approach is quite powerful, it has one major drawback; it requires programmers to write specifications at method boundaries. This has the nice benefit of enabling modular checking, but places an additional burden on the programmer who merely wants to ensure correct protocol usage. Therefore, ANEK has been designed to eliminate this burden by statically inferring the access permission specifications.

We envision programmers using ANEK and PLURAL in the following manner: First, developers of libraries and frameworks would continue to provide PLURAL annotations along with their APIs. This allows the most knowledgeable developers to build the abstractions specific to their APIs, and formally define the ways in which they must be used. Since an API is typically used by many client programs, this effort is amortized over all the users of the API. When a client wishes to use an API (annotated with specifications), however, they start by running the Anek inference tool over their code. Anek will see which API methods are being used and will infer appropriate Plural specifications in the client's code. With the new specifications, the programmer will then run Plural. Since Plural is a sound checker, if Plural passes the resulting program with the newly inferred annotations, it constitutes a guarantee that the programmer is using the API correctly, with little additional burden on the programmer's part.

Anek's inference algorithm is probabilistic. In other words, it solves a number of probabilistic constraints in order to determine what specifications are *likely* to be used (rather than what specifications *must* be used). We have chosen to develop a probabilistic inference, as opposed to a more traditional, logical inference, for a number of reasons. Probabilistic constraints allow us to easily include heuristics, encoding intuitions gleaned from several years of experience writing such specifications. The specifications output by Anek are therefore idiomatic, and in some sense are the most desirable specifications, rather than being just one of many satisfying specifications. But additionally, even the logical constraints, which encode basic invariants of access permissions that should always hold true, are encoded probabilistically. This allows Anek to determine a solution even in the face of conflicting constraints, such as when a bug exists in the program under inference. Because Plural is a sound checker and will be run on the resulting specifications, there is no danger due to the approximation.

On the performance side, there are very real benefits to using an approximate approach. First, it can perform better than an exact approach. But more interestingly, such an approach allows us to infer specifications in a modular way. This is an important goal for Anek since it allows inference to scale to larger programs. In the approach, probabilistic specification summaries describing the current most likely specification are placed at method boundaries. These summaries are refined and made more accurate over time, while still presenting an interface against which method bodies are locally inferred. The details of this algorithm are discussed in Section 3.4.

3. The Anek system

Anek constructs a probabilistic constraint system that is based on logical and heuristic rules from an abstraction of the program under analysis. These constraints are solved in a modular fashion in order to compute the desired specifications.

3.1 Permissions Flow Graph

The inference algorithm performs inference over an abstraction of the program called the *Permissions Flow Graph* (PFG). A PFG is a directed graph of the flow of permissions in each method of a program. Permission flow itself is identical to a data flow except for two differences. First, at method call sites and field assignments, some amount of permission is retained in the calling or assigning context. Second, permission can flow *out* of the arguments at a method call site after the called method returns, representing the manner in which permission is returned to a calling argument at a call site. These are the only two differences.

A PFG G_m for a method m is constructed as follows: For each method parameter, including the receiver for instance methods, two nodes are created. One represents the permission required at the precondition and the other represents the permission returned at the postcondition. The flow of permission that originates at a parameter precondition and is transformed as it passes through a method body is represented using nodes and edges. A control flow graph is constructed in order to determine the flow of the permission. Additionally, a local must-alias analysis helps us track permission (which fundamentally are related to *objects*) even if those objects are reassigned to other local variables. If a variable is passed as an argument to a method, then the graph will contain a directed edge from the previous node in the graph to a node representing the precondition of the corresponding method argument and call site. The entire process is best explained with an example.

```
Row copy(Row original) {
  Iterator<Integer> iter =
    original.createColIter();
  Row result = new Row();
  while( iter.hasNext() ) {
    result.add( iter.next() );
  }
}
```

Figure 5. The copy method from Figure 3 whose representation is given in Figure 6

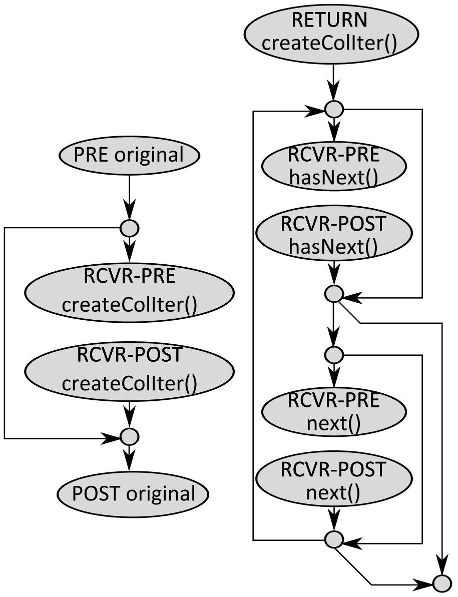

Figure 6. The PFG G_{copy} generated for the method copy in Figure 5.

Consider the method copy shown in Figure 5. This method gives rise to the PFG G_{copy} shown in Figure 6. This figure ignores portions of the graph related to the result variable for simplicity. There are several interesting features worth pointing out.

First, note the path of the permission relating to the original variable, which appears on the left-hand side of the figure. This shows how methods calls are represented when there is no interesting control flow. The node "PRE original" is generated, which corresponds to the permission available to the original variable at the precondition of the copy method. This node is connected to a split node which itself has two successor nodes. The first successor is a node representing the precondition permission to the receiver of the createColIter method. The second successor is a permission merge node. The first edge represents the permission that is passed to the createColIter method, while the second edge represents the permission to the object that is retained by the copy method for the duration of the call. If a permission split is going to occur before the call, where a strong permission is converted to multiple weaker permissions, such as the case when a unique permission is available but only a full permission is needed for a call, it will be manifest at this split node. Next, a node representing the permission to the receiver node returned from the createColIter method is generated. It is important to note that this node has nothing to do with the return value of the `createColIter` method. It merely represents permission to the receiver that is no longer used by the method. The next node, the merge node, combines permission that was held at the call site with any permission returned by the method. Finally, the node "POST original" represents the permission available to the `original` object at the end of the method body.

The right-hand side of Figure 6, which represents the permission returned to the return value of the `createColIter` method, is quite similar. It contains two method calls whose structure is similar to the call to the `createColIter` method. However, because this method call occurs in a loop, there are additional edges to account for the possible control flow paths. For example, an edge connecting the receiver postcondition of the `hasNext` method and the end of the method body exists, and represents the flow of permission to the iterator when the `hasNext` method returns false.

```
Object accessFields(C o) {
  o.f = new Object();
  return o.f;
}
```

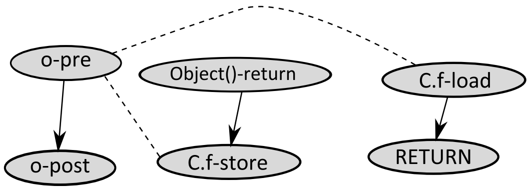

Figure 7. A permission graph containing field accesses and the program from which it was generated. The dotted line represents the reference a field access node maintains to its receiver.

For each field read and each field assignment in a program, a corresponding node is generated in the permission flow graph. Field read nodes will be connected via edges to the variables to which they are assigned, and act as permission sources. Field assignments act as permission sinks. They never contain outgoing edges, but will have incoming edges leading from the nodes from which they are assigned. All instance field read and assignment nodes will separately keep track of the receiver node from which they were accessed. In Figure 7, which illustrates a graph generated from field accesses, the relationship with the receiver node is represented as a dotted line.

3.2 Random variables and prior probabilities

Let $G_m = (V_m, E_m)$ be a PFG for a method m with node set V_m and edge set E_m. We associate each node $n \in V_m$ with five Bernoulli random variables X^n_{unique}, X^n_{full}, $X^n_{immutable}$, X^n_{share} and X^n_{pure}, one for each permission kind. Each variable models the probability that such a permission is available at the associated node. The variables are distributed according to $\mathcal{B}(p)$, a Bernoulli distribution with mean p. If, for instance, X^n_{unique} is distributed according to $\mathcal{B}(0.1)$, this means that node n has permission unique with probability 0.1.

Additionally, for each abstract state in the hierarchy of the type with which the node is associated, we have a random variable associated with that state. For instance, for a node associated with a parameter i where the parameter is an iterator, that node would have three random variables $X^i_{HASNEXT}, X^i_{END}, X^i_{ALIVE}$, one for each of its abstract states.

Each random variable is given a *prior distribution*. This prior distribution models our initial belief of how likely a given variable is to be $true$. For most of the variables Anek creates, we have no information on their respective values (that is, $true$ or $false$), since we are trying to infer the specifications. For that reason, most variables are given a prior distribution equal to $\mathcal{B}(0.5)$ representing this lack of information. However, if a specification already exists in the source program, this strengthens our prior beliefs on how the variables are distributed. For example, consider the specification for the receiver parameter shown in Figure 8. Based on this specification,

```
@Perm(requires="full(this) in HASNEXT",
      ensures="full(this) in ALIVE")
T next() {...}
```

Figure 8. Plural specification for the receiver parameter.

Anek will set the prior distributions for the variables associated with the receiver pre and postcondition nodes. For the precondition node, it will set the prior distributions for both the full permission kind and the "HASNEXT" abstract state to the distribution $\mathcal{B}(0.9)$ denoting the fact that both the full permission and "HASNEXT" abstract state are high probability events. The remaining variables will be given a low prior distribution of $\mathcal{B}(0.1)$. Therefore, the prior distributions for each of the random variables associated with the precondition node are as follows.

Random Variable	Prior Distribution
X_{unique}	$\mathcal{B}(0.1)$
X_{full}	$\mathcal{B}(0.9)$
$X_{immutable}$	$\mathcal{B}(0.1)$
X_{share}	$\mathcal{B}(0.1)$
X_{pure}	$\mathcal{B}(0.1)$
$X_{HASNEXT}$	$\mathcal{B}(0.9)$
X_{END}	$\mathcal{B}(0.1)$
X_{ALIVE}	$\mathcal{B}(0.1)$

It is important to note that even though the specification is given, we still say that the specification permission is only very likely to be $true$ (i.e., $true$ with a probability of 0.9). This allows for the possibility that the original specification was incorrect if the evidence against it (as inferred from the analysis) is overwhelming. Analogous to the nodes, each edge $e \in E_m$ in the PFG is also associated with a similar set of random variables $\{X^e_k\}$, where k is either a permission kind or an abstract state. This allows the distribution of one node to influence the distributions of adjacent nodes via the corresponding edge, as discussed in the next section.

3.3 The probabilistic constraint system

Setting the prior probabilities for the random variables in the PFG models our initial beliefs based on known specifications. Subsequently, we add probabilistic constraints over these random variables that are based on the features of the program itself. These constraints model dependencies that hold among the random variables. A solution to the constraints tells us for each node in PFG, which permission kind and which abstract state are most likely to be true.

We broadly classify all of the constraints that are added as being of one of two types, *logical* constraints or *heuristic* constraints. Logical constraints encode basic permission rules which must always hold, such as those governing sound permission splitting [5]. Heuristic constraints, on the other hand, encode features that are *generally* true of good Plural specifications. Importantly, even though the logical rules must always be true in a program verified by Plural, Anek only dictates that they be true with some high probability. It is precisely this feature which allows Anek to infer specifications even in the face of buggy programs. In the next two sections, we will present each of the constraint generation rules in turn. Each constraint generation rule is parametrized by some probability $h_n \in [0, 1]$ that represents high probability, and is given as input to the algorithm. Parametrization of these high probabilities allows us to tune the performance of inference.

3.3.1 Logical constraints

ANEK encodes the basic logic of access permissions through a series of logical constraints. The logical constraints labeled $L_1, \ldots, L_3$ are described below.

L_1***: Outgoing Permissions*** At every node in the graph, the permission at the node must be related somehow to the permissions on its outgoing edges. If the node only has one outgoing edge, this is pretty easy. The edge and the node must have the exact same permission. So the following constraint is applied on the permission and state random variables associated with that node (n) and edge (e), with high probability:

$$\bigwedge_{k \in \{\textsf{unique},\textsf{full},\textsf{immutable},\textsf{share},\textsf{pure}\} \cup \textsf{states}(n)} X_k^n = X_k^e \,|_{h_1} \qquad (1)$$

However, if there are multiple outgoing edges, the story is a little more complicated. Nodes can have multiple outgoing edges for one of two reasons. In some cases, it is because the permission at the node is being split into multiple permissions as new aliases are introduced. In other cases, the multiple edges are due to control flow branches in the original program. Since permission splits can only occur before method calls and field reads, we can mark these nodes as such, and apply different rules for splitting and control flow branches. At branches, we apply constraint 1 for each outgoing edge, mandating that the permission available at the node is equal to the permission available at each outgoing edge. For permission splits, however, the permission on the outgoing edges generally cannot be identical.

There are certain ways in which an access permission can be soundly split. For example, a unique permission can be split into two share permissions, two immutable permissions or two pure permissions. It cannot, however, be split into two full permissions or two unique permissions, as those two newly created permissions would violate the assumptions made by one another. Therefore, at each node, constraints are placed on the corresponding random variables that will enforce sound permission splitting. The (long) series of constraints is as follows:

$$\begin{array}{l}
((X_{unique}^n \wedge \bigwedge_{e \in \textsf{outgoing}(n)} \bigvee_{k \in \{\textsf{unique},\textsf{full},\textsf{imm},\textsf{share},\textsf{pure}\}} X_k^e) \vee \\
(X_{full}^n \wedge \bigwedge_{e \in \textsf{outgoing}(n)} \bigvee_{k \in \{\textsf{full},\textsf{imm},\textsf{share},\textsf{pure}\}} X_k^e) \vee \\
(X_{imm}^n \wedge \bigwedge_{e \in \textsf{outgoing}(n)} \bigvee_{k \in \{\textsf{imm},\textsf{share},\textsf{pure}\}} X_k^e) \vee \\
(X_{share}^n \wedge \bigwedge_{e \in \textsf{outgoing}(n)} \bigvee_{k \in \{\textsf{share},\textsf{pure}\}} X_k^e) \vee \\
(X_{pure}^n \wedge \bigwedge_{e \in \textsf{outgoing}(n)} X_{pure}^e)) \wedge \\
\bigwedge_{e \in \textsf{outgoing}(n)} X_{unique}^e \Rightarrow \bigwedge_{e_2 \in \textsf{outgoing}(n) - e} \neg (X_{unique}^{e_2} \vee X_{full}^{e_2}) \\
\bigwedge_{e \in \textsf{outgoing}(n)} \bigwedge_{s \in \textsf{states}(n)} X_s^e = X_s^n \,|_{h_2}
\end{array} \qquad (2)$$

L_2***: Incoming Permissions*** When a node has incoming edges, ANEK also adds constraints on the relationship between the incoming edge permissions and the node permission. Specifically, ANEK says that the permission associated with a node is equal to one of the permissions on the incoming edges with high probability. The following constraints are generated for a node n with incoming edges:

$$\bigvee_{e \in \textsf{incoming}(n)} \bigwedge_{k \in \{\textsf{unique},\textsf{share},\textsf{imm},\textsf{share},\textsf{pure}\} \cup \textsf{states}(n)} X_k^n = X_k^e \,|_{h_3} \qquad (3)$$

L_3***: Field Write*** For any field store node (i.e., field assignment), the associated receiver node cannot be associated with one of the read-only permissions, immutable or pure. This constraint then sets the receiver to be immutable or pure with a very low probability. A field cannot be modified without writing permission to its receiver, so whenever we see a field store we know that we have writing permission to the receiver object.

3.3.2 Heuristic Constraints

For the random variables generated from a PFG, a series of additional "heuristic" constraints are also added. These constraints correspond to our intuitions about what makes a good PLURAL specification.

H_1***: Constructors*** Constructors generally return unique permission, so for the specification for the object created by a constructor, we say that the variable X_{unique} is likely to be true with elevated probability. This is merely a heuristic since constructors do not have to return unique permission. Aliases can be introduced and stored in various data structures before the constructor returns.

H_2***: Pre and Post*** For a given parameter of a method, the permission kind, but not the state, of the pre and postcondition nodes are the same with high probability. This is again a heuristic as it is possible for methods to retain an input permission and return a different permission.

H_3***: Factory Methods*** Methods whose names begin with the word "create" usually return a unique permission, much like a constructor. These methods in practice are often static factory methods. Therefore, in our analysis, the return variable from such methods, X_{unique}, is true with elevated probability.

H_4***: Setter Methods*** Methods whose names begin with the word "set" generally require a writing permission (i.e., unique, full or share) to their receiver, since they are often used to write to receiver fields. Therefore, when encountering such a method in the PFG, the variables $X_{immutable}$ and X_{pure} for the receiver pre and postcondition are constrained to be true with low probability.

H_5***: Thread-Shared*** Targets of synchronized blocks are of full, share or pure permission with high probability. This heuristic is based on ideas developed in the concurrent version of the PLURAL analysis [3]. The permissions full, share and pure are the three permission kinds that may indicate possible thread-shared objects.

3.4 Probabilistic model and inference

In the previous section, we described how logical constraints L_1, L_2, L_3 and heuristic constraints H_1, H_2, H_3, H_4, H_5 can be derived from a PFG representation. We will now show how these constraints can be looked upon as a probabilistic model, in particular, a joint probability distribution describing the whole space of specifications.

Let $X_1, \ldots, X_n$ be n Bernoulli random variables where for each $1 \leq i \leq n$, X_i takes values from the domain $\{0, 1\}$. Let $p(X_1, \ldots, X_n)$ be a joint probability distribution function over these variables. Associated with $p(X_1, \ldots, X_N)$ are n *marginal functions* $p_i(X_i), 1 \leq i \leq n$ defined as:

$$\begin{array}{l} p_i(X_i) = \\ \displaystyle\sum_{X_1 \in \{0,1\}} \cdots \sum_{X_{i-1} \in \{0,1\}} \sum_{X_{i+1} \in \{0,1\}} \cdots \sum_{X_n \in \{0,1\}} p(X_1, \ldots, X_n) \end{array} \qquad (4)$$

where the sum is over all variables except X_i. Intuitively, the marginal function $p_i(X_i = a)$ corresponds to the probability of the variable X_i taking the value $a \in \{0, 1\}$. Since there are an exponential number of terms in the above equation, a naïve algorithm for computing $p_i(X_i)$ is not tractable. Suppose that $p(X_1, \ldots, X_n)$ can be written as a product of functions, where each function has some subset of $\{X_1, \ldots, X_n\}$ as its arguments, that is:

$$p(X_1, \ldots, X_n) = \prod_{j \in J} f_j(\mathcal{Z}_j) \qquad (5)$$

where J is a discrete index set, $\mathcal{Z}_j \subseteq \{X_1, \ldots, X_n\}$ and $f_j(\mathcal{Z}_j)$ is a function having the elements of $\mathcal{Z}_j$ as arguments, with the interval $(0, 1]$ as its range, and the product is the usual pointwise product of functions. Given this factorization, there are a number of machine learning techniques that efficiently estimate the marginal functions by exploiting the fact that every factor is a function of small number of variables [14].

In our setting, each of the functions $f_j(\mathcal{Z}_j)$ in Equation 5 is a probabilistic constraint that describes either a logical constraint or a heuristic constraint as defined in Section 3.3. For instance, consider the conjunct $X^n_{share} = X^e_{share} \mid_{h_1}$ in Equation 1 (logical constraint L_1). This can be encoded as a probabilistic constraint as follows.

$$f(X^n_{share}, X^e_{share}) = \begin{cases} h_1 & \text{if } (X^n_{share} = X^e_{share}) \\ 1 - h_1 & \text{otherwise} \end{cases} \qquad (6)$$

The pointwise product of such probabilistic constraints represents the probability space over specifications. The specifications are computed by sampling the marginal functions (defined by Equation 4) for the joint probability distribution. The problem of inferring specifications can be formally defined as follows:

Definition 1. *Let P be a program defined by a set of methods M, and let Π_m be the probabilistic model for a method $m \in M$ defined as follows.*

$$\Pi_m = \Gamma_m \cdot \left(\prod_{c \in \text{CALLSITES}(m)} \text{PARAMARG}(c) \right)$$

where Γ_m denotes the pointwise product of the logical and heuristic constraints associated with method m, and PARAMARG*(c) defines a set of equality constraints that bind the method m's parameters to their respective arguments at a call site c. The probabilistic model Π_P for the program P is the product of the probabilistic models for all its methods and defined as.*

$$\Pi_P = \prod_{m \in M} \Pi_m$$

Then, we are interested in computing marginal functions for the probabilistic model Π_P.

It is important to note that the specifications for the program can be easily derived from the marginal functions of Π_P via sampling. These marginal functions can be computed by using an off-the-shelf machine learning algorithm. However, a serious drawback of such an approach is that it would not be modular, thus severely limiting scalability.

We propose an inference procedure ANEK-INFER that is a modular analysis (shown in Figure 9). The set of all methods M in the program under analysis is the input to this procedure. $\mathcal{X}_m$ is the set of all random variables for a method m as defined in Section 3.2. We denote the PFG for method m by G_m. The probabilistic constraints system Γ_m is as defined in Definition 1.

In line 1, a worklist W of probabilistic models is initialized to the empty list. Next, in lines 2–6, for every method m:

(a) The random variables in $\mathcal{X}_m$ are initialized (described in Section 3.2).

(b) A probabilistic constraint system Γ_m is created by the procedure Model(G_m).

```
procedure ANEK-INFER
 input:
   $M$: Set of all methods in the program
 vars:
   $\mathcal{X}_m$: Set of all random variables for method $m$
   $G_m$: PFG for method $m$
   $\Gamma_m$: Probabilistic constraints for method $m$
   $W$: Worklist of probabilistic models
   $\pi_m$: Probabilistic summary for method $m$
   $\sigma_m$: A deterministic summary for method $m$
 output:
   $\sigma$: Specification that maps every method to its deterministic
   summary
 1: $W := \emptyset$
 2: for each $m \in M$ do
 3:   INIT($\mathcal{X}_m$)
 4:   $\Gamma_m$ := MODEL($G_m$)
 5:   $W := W \cup \{\Gamma_m\}$
 6: end for
 7: $count := 0$
 8: while $count \leq MaxIters$ do
 9:   $count := count + 1$
10:   $\Gamma_m$ := CHOOSE($W$)
11:   $W := W \setminus \{\Gamma_m\}$
12:   for each $c \in$ CALLSITES($m$) do
13:     APPLYSUMMARY($\Gamma_m$, $\pi_c$))
14:   end for
15:   $\mathcal{X}^{old}_m := \mathcal{X}_m$
16:   $\mathcal{X}_m$ := SOLVE($\Gamma_m$)
17:   if $\mathcal{X}_m \neq \mathcal{X}^{old}_m$ then
18:     UPDATESUMMARY($\pi_m$, $\mathcal{X}_m$)
19:     $W := W \cup \{\Gamma_m\}$
20:   end if
21: end while
22: for each $m \in M$ do
23:   for each $X \in \pi_m$ do
24:     if $p(X = true) > t$ then
25:       $\sigma_m(X) := true$
26:     end if
27:   end for
28:   $\sigma(m) := \sigma_m$
29: end for
30: return $\sigma$
```

Figure 9. The annotation inference algorithm ANEK-INFER.

(c) The worklist W is initialized to a list of probabilistic models for methods.

ANEK-INFER is an iterative algorithm (lines 8–21). In lines 10 and 11, a model Γ_m for method m is picked from the worklist W by the choice function CHOOSE(W) and removed from W. In lines 12–14, for every call site corresponding to a method c, a probabilistic summary π_c for that method is applied to the model Γ_m (where method m calls method c). A *probabilistic summary* π_c for method c is the set of all random variables associated with the precondition and postcondition nodes in its PFG G_c. Due to the initialization in line 3, all probabilistic summaries for methods are also appropriately initialized. A probabilistic summary maintains the current values of the random variables corresponding to the precondition and postcondition nodes for a method. The procedure Solve called in line 16 takes Γ_m as input and computes approximate

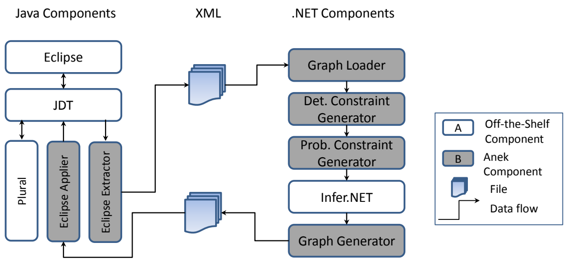

Figure 10. The architecture diagram for Anek.

marginal functions[3] for the variables $\mathcal{X}_m$. We use an off-the-shelf machine learning algorithm to efficiently implement Solve. If the distributions associated with the random variables in $\mathcal{X}_m$ have changed, then the summary π_m for method m is updated via the procedure UpdateSummary (line 18) and the model Γ_m is added to the worklist W (line 19). It is important to note that summaries are computed by analyzing each method once at a time and it is via these summaries that analysis information is disseminated across methods. This makes the Anek-Infer algorithm a modular analysis.

The loop (8–21) is run $MaxIters$ number of times (as opposed to computing a fixpoint) which is another source of approximation. Lines 22–27 compute the specification from the probabilistic summaries for each procedure. If for each random variable X in the probabilistic summary π_m for a method m, the probability $p(X = true)$ is greater than a user-defined threshold $t \in [0.5, 1)$, then the deterministic value of that variable is set to $true$ and this information is stored in a map σ_m. This deterministic summary σ for all methods forms the specification that is returned by Anek-Infer. It is interesting to note that the result computed by Anek-Infer(M) at a fixpoint (corresponding to an exact computation) is identical to the result computed by Solve(Π_P).

4. Evaluation

In this section, we describe our implementation and report the empirical results of running the combination of Anek and Plural on a number of Java benchmark programs. All experiments were performed on a system with a 3.19 GHz Intel Pentium 4 processor and 2 GB RAM running Microsoft Windows XP.

4.1 The Anek Implementation

The architecture of Anek is shown in Figure 10. Each component of Anek is organized into a pipeline. The first component, the Eclipse Extractor, is a plugin to the Eclipse Java Development Tools (JDT), the Java IDE in Eclipse. Its job is to visit the Java AST generated by JDT for the program under inference and generate the abstract representation. This representation is stored to disk in an intermediate, XML-based format. This component is also responsible for the user interface, the menus and action items in Eclipse that allow users of Plural to run inference without ever leaving their IDE.

Once Anek has generated an XML representation of the program, this representation is handed off to an F# program[4]. It begins by generating a number of constraints based on the shape of the program (Deterministic Constraint Generator). These are the constraints described in Section 3.3. However, at this phase each of the constraints is deterministic; there are no probabilities involved.

It is in the next component, the Probabilistic Constraint Generator, that the deterministic constraints are transformed into probabilistic ones and the Anek-Infer algorithm is invoked. Anek-Infer is implemented using Infer.NET [16], a library that exposes a number of abstractions, in the form of types and methods, which allow networks of probabilistic variables and constraints to be assembled and solved. If the most likely permission kind and abstract state, as determined by Anek-Infer, is greater than some threshold value, then a generated graph representation will contain this newly inferred specification.

Finally, in the last component, the newly generated representation is applied to the original Java program inside another Eclipse plugin called the Eclipse Applier. This program walks through the AST of the original Java program and applies the new specification.

4.2 Experiments

In order to evaluate the utility of Anek, we performed a number of small experiments and one main experiment. The goal was to see if Anek would infer annotations that were correct and would not lead to a large number of false warnings when running the Plural tool.

First, we developed a number of test benchmarks. Each of these benchmarks consisted of one or more classes, with one or more methods, some of which were annotated by us before running the Anek tool. Each experiment was designed to test some particular Anek constraint or feature. During the experiments we would run Anek on the test suite, and ensure that correct annotations were inferred, and that after inference Plural would report no warnings. At issue is the evolution of Anek in response to newly perceived problems. One of the great benefits of Anek's architecture is that it is so easy to evolve it by adding new constraints. Over the course of its implementation we added new constraints or modified existing constraints numerous times. However, we wanted to ensure that the new constraints, which may fix one particular problem, did not come at the expense of any previously-correct behavior. Therefore, our small experiment suite formed a regression suite of sorts and also as a training set to fine-tune the parameters of the inference engine.

Our primary experiment, though, was to use Anek to infer annotations for the PMD static analysis framework (Table 1). In this experiment, the Java Iterator API was annotated and then Anek was used to infer annotations within the PMD application, which makes extensive use of the API.

PMD	
Lines of Source:	38,483
Number of Classes:	463
Number of Methods:	3,120
Calls to Iterator.next():	170

Table 1. Simple statistics for the PMD application.

Specifically, PMD was used as the client-side case study in Kevin Bierhoff's doctoral thesis [4]. In his experiment, Bierhoff took an annotated Iterator API, ran Plural on PMD, and added

[3] Since we are only interested in estimating the likely values of the random variables and not their exact distributions, computing approximate marginal functions suffices for our purpose.

[4] `http://research.microsoft.com/fsharp/fsharp.aspx`

appropriate annotations **by hand** to the program until there were as few remaining warnings as possible. Our goal was essentially to replicate this experiment by using ANEK instead of doing any specification by hand.

Method	Annotations	Warnings	Time Taken
Original	0	45	0
Bierhoff	26	3	75min [4]
Anek	31	4	3min 47sec
Anek Logical	N/A	N/A	DNF

Table 2. The results of running ANEK on PMD.

Table 1 contains a number of basic statistics for the PMD application. Of particular note is the number of calls to the method, `java.util.Iterator.next`. This is important because the `next` method is the most important for verification purposes. It is the only method on the `Iterator` interface that requires the iterator instance to be in a particular state when called.

Table 2 shows the results of our experiments. We ran several experiments and recorded three statistics for each one. The first configuration is Original, where we ran PLURAL on PMD with no annotations at all, in its original form. The point of this experiment is just to show that *some* annotations must be inferred in the application in order to verify correct use of the Iterator API. To that end, PLURAL reported 45 warnings when run on the unannotated program. The next configuration, Bierhoff, is PMD as annotated by Bierhoff for his thesis work. Manual annotation took 75 minutes as reported in [4]. PLURAL reported three warnings, all of which were false positives. In these three cases, the `next` is called on an iterator without first calling the `hasNext` method to establish dynamically that the iterator has subsequent elements. In all three cases, other program invariants not expressed in PLURAL guarantee that the call to `next` will not fail at run-time because the underlying collection is known to be non-empty. In fact, these cases were quite similar to, and the insipration for the `testParseCSV` method from Figure 3.

The next experiment uses ANEK to infer annotations on PMD. The Anek configuration is the standard configuration. When running PLURAL on PMD with the annotations inferred by ANEK, four warnings are generated, and the inference process takes 3 minutes and 47 seconds. Of the four warnings, three are exactly the same warnings issued by PLURAL as in the Bierhoff configuration. The fourth warning, which is also a false-positive, can be attributed to a lack of branch-sensitivity. While the PLURAL static analysis takes the result of conditional expressions into account, ANEK currently does not, and therefore cannot infer the correct specification for a method that is only called in $true$ branches of a conditional. As is evident from the experiments, ANEK performs as well as with hand-coded annotations, and at approximately 5% of the elapsed time and with no human involvement.

Finally, in addition to comparing ANEK to manual annotation, we wanted to compare it to a more traditional specification inference approach, specifically, approaches that are built on top of logical constraint solvers. While such an inference tool currently does not exist for PLURAL, two experiments were performed in an attempt to understand how such tools might perform.

First, we modified ANEK to add an additional Logical mode. In this configuration only logical constraints are considered and solved deterministically, while all heuristic constraints are turned off. In this experiment, Anek Logical was run on PMD in an attempt to infer specifications. When this was done, the inference procedure ran out of memory before a fixed point was reached, and therefore no results could be presented.

Our second attempt to compare ANEK to a traditional inference algorithm relies on PLURAL's own local permission inference.

Inference Tool	Time Taken	Warnings
ANEK	22 sec	0
Plural Local Inference	181 sec	0

Table 3. The results of running ANEK on a test program to compare performance again Plural's local inference

While PLURAL requires annotations on method boundaries it uses a local permission inference so that programmers do not have to write annotations on local variables. This analysis is responsible for determining which fractions of permissions are consumed and returned by different parts of a method body, for finding a satisfying assignment for all of the various permission constraints imposed by all of the called methods and returned permissions. The underlying algorithm relies upon Gaussian Elimination to find satisfying fractional permission assignments [4, ch. 5]. The overall approach is comparable to other similar fractional inference algorithms [19]. In order to use PLURAL's local inference as a point of comparison, we took a small test program crafted for this experiment which contained numerous short methods and ran ANEK on it to infer method specifications. Then, in a second run, we inlined each method so that the resulting program consists of one single large method and ran PLURAL on this program. Both inference tools end up doing the same work, since by solving permission constraints for the entire large method body, PLURAL is essentially inferring which permissions must be available at the same points as ANEK. Table 3 shows the results for this experiment. The program under inference is small (400 lines) but contains numerous control flow branches. ANEK performed well, doing the same inference task in roughly one ninth of the time.

4.3 Discussion

In this section, we will discuss the results of our experiment and their ramifications. Overall, we were quite pleased with the results of the experiment. In approximately 5% of the time it took to annotate the program by hand, ANEK was able to infer specifications that were almost as good, and with no human involvement. Specifically, the specifications inferred by ANEK lead to four warnings when the PLURAL tool was subsequently run on the result, versus three warnings from hand-written specifications. This difference of one warning is entirely due to ANEK's lack of path-sensitivity in its inference scheme, a feature PLURAL itself supports. Precision-wise, ANEK does a very good job with the annotations it infers. Moreover, when compared with other PLURAL case studies where annotations were applied manually [6], the PMD case study went very quickly. Often it takes several hours to manually annotate a code base, so we expect to see even bigger time savings on future experiments.

While our experiments comparing ANEK to logical inference tools were somewhat rough, they suggest that ANEK can outperform traditional-style inference tools. We largely attribute this to the use of approximation algorithms for probabilistic constraint solving.

We also claimed that our approach is a good idea because probabilistic permissions enable reasoning in the face of conflicting constraints. This feature was needed for our PMD experiment because of the three locations in which the `next` method was called without a proceeding call to `hasNext`. Just as in the example presented in the introduction, the conflicting constraints introduced by such calls were tolerated, and satisfactory specifications were still inferred rather than ANEK giving up. Given that the remaining 167 calls to the `next()` method were correctly verified by PLURAL, the resulting specifications are still quite useful to programmers.

Description	Count
Same	14
ANEK Added Helpful Spec.	6
ANEK Added Constraining Spec.	1
ANEK Removed Spec.	3
ANEK Changed Spec., More Restrictive	6
ANEK Changed Spec., Wrong	3

Table 4. Comparison of by-hand annotations with Anek

The quality of the specifications inferred by ANEK is generally good. Table 4 summarizes the annotations inferred. Specifically, these numbers are for the specifications inferred by ANEK with respect to the hand-specified version (Bierhoff). 14 of the specifications were exactly the same. ANEK inferred 6 specifications that were correct, potentially useful in future versions of the application and imposed no additional proof burden. In one case ANEK added a specification that was not necessary and may, in the future, cause additional proof burdens. In 3 cases, ANEK did not infer a specification that was present in the hand-specified version. All three of these were related to dynamic state test methods, which ANEK currently does not attempt to infer. The removed specifications were immaterial because at all use sites, a super-type specification took precedence. In 6 places, ANEK changed an existing specification to make it more restrictive, which, while not causing any additional errors now, may lead to additional proof burdens in future versions of the application. Finally, 3 specifications were wrong outright. One of these incorrect specifications led to the additional warning. The other two did not affect verification at all.

One nice benefit of creating an analysis based on probabilistic constraints is the ease of design. It turned out to be quite easy to add new constraints. As we went through our design iterations, we started with a basic suite of probabilistic constraints that we thought would yield good results. However, on some of our small benchmarks, we found that for one reason or another these constraints were not quite yielding the expected results. Fortunately, it is quite easy to add new constraints. So, as we realized that one constraint was overly specific, or that another, say, did not work in all situations, it was trivial to add a new constraint so that the results would be more to our liking.

5. Related work

Our work is related to MERLIN [15], a tool based on probabilistic analysis for inferring security annotations useful for detecting information flow vulnerabilities. In general, the specifications ANEK infers are much more detailed and intricate behavioral properties than those inferred by MERLIN. Furthermore, ANEK introduces a general framework that is based on the philosophy of combining logical rules with heuristic rules. As such, ANEK must know much more about the details of each function's behavior. In contrast, MERLIN's annotation inference is based on how functions are used in the implementation and does not rely on the function's actual behavior. Moreover, the inference in MERLIN is not modular and this limits its scalability. Additionally, with ANEK, we have the nice feature that after specifications are inferred, a sound static analysis can be run, verifying the results of the inference and acting as a safety net. For MERLIN, such a tool was unavailable.

Kremenek et al. [13] propose a technique for inferring ownership annotations that is also based on a probabilistic analysis. Like MERLIN, this analysis also infers less expressive specification and is not modular.

Dietl [9] developed a global analysis for inferring Universe Type annotations using a SAT solver. Besides the fact that we are inferring typestate annotations, the primary difference between this work and ours is that it requires satisfiability. If a program has bugs and therefore has no valid ownership type, the inference will fail with unsatisfiable constraints. On the other hand, ANEK will always produce the best possible specification.

Terauchi [19] proposed a global analysis for inferring fractional permissions in order to verify a lack of race conditions. While the methodology itself solves a problem that might be useful in our work, their underlying methodology is much different, since they do not use probabilistic constraints. Presumably such an analysis would have to give up when confronted with false positives of the sort we encountered in our case study.

Houdini [10] is an annotation inference engine for the ESC/Java tool [12]. It first heuristically generates a number of candidate annotations or invariants and subsequently, incorrect annotations are pruned away by invoking ESC/Java. This is similar to the manner in which the specifications inferred by ANEK are checked by PLURAL in order to ensure soundness of verification.

An important line of work has addressed the related but different problem of *protocol inference*. Such approaches have used static [1, 17, 20] and dynamic [21, 22] analysis to determine which classes in a program place restrictions on the ordering of their method calls, and attempt to infer a specification for their correct usage. The dynamic approaches generally use statistical methods to determine which sequences of method calls represent protocols, and which are merely due to coincidence. While the problem of inferring protocols is different from the problem of inferring aliasing annotations, these approaches clearly complement our own, and in the future we plan to investigate their combination.

6. Conclusion

In this paper we presented ANEK, a probabilistic specification inference tool that can be used to infer access permissions for use in modular typestate checking. ANEK is novel in that it is modular and is built using probabilistic constraints. These constraints allow us as developers to easily encode into the analysis our understanding of what makes a good specification. Probabilistic constraints also make ANEK robust to bugs in the program under inference and enable a modular analysis. In order to evaluate our approach, we used ANEK to infer specifications for PMD, mimicking a case study that was performed by Bierhoff [4] by hand as part of his Ph.D. thesis. The results were good – in fact, the specifications inferred by ANEK were nearly as good as those written by hand and were obtained in approximately 5% of the time it took to manually discover them.

Acknowledgments

We thank G. Ramalingam for suggesting the idea of classifying ANEK's constraints as logical and heuristic constraints. We also thank Jonathan Aldrich, Akash Lal and Sriram Rajamani for their invaluable comments. Thanks are also due to John Winn for help with using the INFER.NET API. The first author was supported by several grants including DARPA grant #HR0011-0710019, NSF grant CCF-0811592, R&D Project Aeminium CMU-PT/SE/0038/2008 in the CMUPortugal program, Army Research Office grant #DAAD19-02-1-0389 entitled "Perpetually Available and Secure Information Systems," the Department of Defense, the Software Industry Center at CMU and its sponsors, especially the Alfred P. Sloan Foundation and a National Science Foundation Graduate Research Fellowship (DGE-0234630).

References

[1] R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *POPL '05: Principles of Programming Languages*, pages 98–109, 2005.

[2] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *PASTE '05: Program Analysis For Software Tools and Engineering*, pages 82–87, 2005.

[3] N. E. Beckman, K. Bierhoff, and J. Aldrich. Verifying correct usage of atomic blocks and typestate. In *OOPSLA '08: Object Oriented Programming Systems, Languages, and Applications*, pages 227–244, 2007.

[4] K. Bierhoff. *API Protocol Compliance in Object-Oriented Software*. PhD thesis, Carnegie Mellon University, April 2009.

[5] K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *OOPSLA '07: Object Oriented Programming Systems, Languages and Applications*, pages 301–320, 2007.

[6] K. Bierhoff, N. E. Beckman, and J. Aldrich. Practical API protocol checking with access permissions. In *ECOOP '09: European Conference on Object-Oriented Programming*, pages 195–219, July 2009.

[7] J. Boyland. Checking interference with fractional permissions. In *SAS '03: Static Analysis Symposium*, pages 55–72. Springer, 2003.

[8] J. Condit, B. Hackett, S. K. Lahiri, and S. Qadeer. Unifying type checking and property checking for low-level code. In *POPL '09: Principles of Programming Languages*, pages 302–314, 2009.

[9] W. Dietl. *Universe Types: Topology, Encapsulation, Genericity, and Tools*. PhD thesis, ETH Zurich, December 2009.

[10] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *FME '01: International Symposium of Formal Methods Europe*, pages 500–517, 2001.

[11] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *PLDI '02: Programming Language Design and Implementation*, pages 234–245, 2002.

[12] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI '02: Programming Language Design and Implementation*, pages 234–245, 2002.

[13] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: Inferring the specification within. In *OSDI '06: Operating Systems Design and Implementation*, pages 161–176, 2006.

[14] F. R. Kschischang, B. J. Frey, and H. A. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 2(47):498–519, 2001.

[15] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee. Merlin: specification inference for explicit information flow problems. In *PLDI '09: Programming Language Design and Implementation*, pages 75–86, 2009.

[16] T. Minka, J. Winn, J. Guiver, and D. Knowles. Infer.NET 2.4, 2010. Microsoft Research Cambridge. http://research.microsoft.com/infernet.

[17] M. K. Ramanathan, A. Grama, and S. Jagannathan. Static specification inference using predicate mining. In *PLDI '07: Programming Language Design and Implementation*, pages 123–134, 2007.

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

[19] T. Terauchi. Checking race freedom via linear programming. In *PLDI '08: Programming Language Design and Implementation*, pages 1–10, 2008.

[20] J. Whaley, , M. C. Martin, , and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *ISSTA '02: International Symposium on Software Testing and Analysis*, pages 218–228, 2002.

[21] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: Mining temporal API rules from imperfect traces. In *ICSE '06: International Conference on Software engineering*, pages 282–291, 2006.

[22] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending API usage patterns. In *ECOOP '09: European Conference on Object-Oriented Programming*, pages 318–343, 2009.