

Symbolic Mining of Temporal Specifications*

Mark Gabel Zhendong Su

Department of Computer Science
University of California, Davis
{mggabel,su}@ucdavis.edu

ABSTRACT

Program specifications are important in many phases of the software development process, but they are often omitted or incomplete. An important class of specifications takes the form of temporal properties that prescribe proper usage of components of a software system. Recent work has focused on the automated inference of temporal specifications from the static or runtime behavior of programs. Many techniques match a specification pattern (represented by a finite state automaton) to all possible combinations of program components and enumerate the possible matches. Such approaches suffer from high space complexity and have not scaled beyond simple, two-letter alternating patterns (e.g. $(ab)^*$). In this paper, we precisely define this form of specification mining and show that its general form is NP-complete.

We observe a great deal of regularity in the representation and tracking of all possible combinations of system components. This motivates us to introduce a symbolic algorithm, based on binary decision diagrams (BDDs), that exploits this regularity. Our results show that this symbolic approach expands the tractability of this problem by orders of magnitude in both time and space. It enables us to mine more complex specifications, such as the common three-letter resource acquisition, usage, and release pattern $((ab^+c)^*)$. We have implemented our algorithm in a practical tool and used it to find significant specifications in real systems, including Apache Ant and Hibernate. We then used these specifications to find previously unknown bugs.

Categories and Subject Descriptors

D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*; F.3.1 [Logics and Meaning of Programs]: Specifying and Verifying and Reasoning about Programs

*This research was supported in part by NSF CAREER Grant No. 0546844, NSF CyberTrust Grant No. 0627749, NSF CCF Grant No. 0702622, US Air Force under grant FA9550-07-1-0532, and a generous gift from Intel. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICSE '08, May 10–18, 2008, Leipzig, Germany.

Copyright 2008 ACM 978-1-60558-079-1/08/05 ...\$5.00.

General Terms

Languages, Algorithms, Verification

Keywords

Specification mining, dynamic analysis, formal specifications

1. INTRODUCTION

Accurate specifications are invaluable assets for the software development process. These specifications often take the form of finite automata that describe *temporal properties*. These properties formally specify the legal interaction patterns with specific components of a system. Canonical examples include alternation properties, like lock acquisition and release, and resource ownership properties, like the requirement that calls to `read(fd)` must all exist between calls to `open(fd)` and `close(fd)`.

When used prescriptively, temporal specifications can help to inexpensively prevent future bugs. When used retroactively, these specifications are candidates for verification by model checkers and static verifiers [3, 9, 12, 24], lighter weight static analysis techniques [7, 14], or manual inspection. These properties can also be verified through software testing, where they can be verified as system invariants. The characteristics of these properties can aid in the generation of test cases.

These specifications are often missing, incorrect, or otherwise incomplete. Researchers have recognized this problem and have sought to develop methods for retroactively *inferring* specifications from systems based on their common behavior. Current techniques can be categorized by the inputs and outputs of their respective algorithms. Most algorithms take as input some type of program or program trace and produce one or more temporal specifications as finite automata. More specifically, some techniques [1, 2, 23] seek to produce a single, arbitrarily complex automaton based on a known alphabet. For example, one might wish to learn the interaction patterns of a language's socket API. The user provides the algorithm with the API functions and a program representation (either the source code or dynamic traces), and a finite automaton describing the probable correct behavior is returned.

Other techniques [10, 18, 21, 25] take as input a specification *pattern* in the form of an automaton. The user provides the algorithm with a larger set of input symbols, often representing every function in the system or event in the trace, and a program representation. The algorithm then enumerates the set of assignments of the input symbols into the automaton's alphabet such that the automaton's interaction pattern holds over the program. Most previous work has focused on *alternating* patterns. For example, the user can provide an alternating template representing the regular expression $(ab)^*$, a program, and an alphabet of possible assignments. The

algorithm returns a set of properties that match this expression, including properties like lock acquisition and release.

One possible reason for this limitation to alternating patterns is the expense of tracking all possible assignments into the input automaton's alphabet. Basic algorithms track the state of every possible assignment over the program's execution. This approach has a large space requirement: for n input symbols and a pattern alphabet of size k , these algorithms require $O(n^k)$ space. Because there are only two participants in an alternating pattern, the computation can be performed in quadratic space.

This work provides a novel symbolic technique for this type of specification inference that is efficiently implemented using binary decision diagrams (BDDs) [6]. Our technique reduces both the space and time requirements by using the regularity of the problem space to form an efficient symbolic representation.

Under empirical evaluation on real traces from large systems, we observe that our algorithm is able to extend this approach to specification patterns with alphabets of at least three letters. More importantly, this allows the inference of the previously mentioned common resource acquisition/use/release protocol, which is described by the pattern $(ab^+c)^*$. We have applied our algorithm to traces from real systems and used the discovered properties to find previously unknown bugs.

This paper makes the following technical contributions:

1. We formalize the pattern-based specification mining problem and show that its general form is NP-Complete.
2. We provide a symbolic algorithm based on *Binary Decision Diagrams* (BDDs) that greatly expands the computational tractability of this problem.
3. We provide practical modifications to our algorithm to allow further scaling and tolerance of imperfect input data.
4. We implement a specification mining tool based on our algorithm and evaluate its performance. We use this tool to mine several interesting specifications, which we use to locate previously unknown bugs in real systems.

The rest of this paper is structured as follows. We begin with a formal definition of the pattern-based specification mining problem and an analysis of its computational complexity (Section 2). We continue with the presentation of our symbolic algorithm (Section 3), which includes the necessary background information on BDDs. We then discuss our implementation and present the results of our empirical evaluation (Section 4). Finally, we discuss related work (Section 5) and conclude with ideas for future work (Section 6).

2. FORMAL ANALYSIS

In this section, we formalize the pattern-based specification mining problem and analyze its computational complexity. We also analyze its expressiveness to motivate our work.

2.1 The Specification Mining Problem

Our work focuses on mining properties from dynamic traces and is most related to the work of Yang *et al.* [25]. We begin with a precise definition of the problem.

Definition 2.1 (Specification Mining) *Assume we have a specification template represented by an automaton A over an alphabet Σ and an execution trace T over a disjoint alphabet Σ' , i.e., $\Sigma \cap \Sigma' = \emptyset$. The pattern-based specification mining problem is defined as the enumeration of all satisfying total and one-to-one*

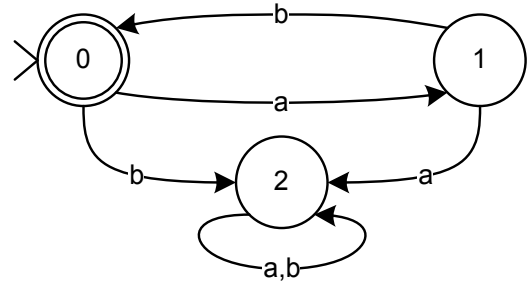


Figure 1: An automaton representing the alternating pattern.

mappings $\rho : \Sigma \rightarrow \Sigma'$ such that the projected trace of T over the range of ρ is accepted by $\rho(A)$, which is the same as A except each transition $\delta(s, a) = t$ is replaced by $\delta(s, \rho(a)) = t$.

Consider this example trace:

open, use, use, close, dispose, get, get, open, use, close, dispose

Suppose we mine the *alternating* automaton pattern in Figure 1 over this trace. In this example, the assignments (a =open, b =close), (a =open, b =dispose), and (a =close, b =dispose) are satisfied over the trace.

To demonstrate the complexity of this problem, we show that its general, decision problem counterpart is NP-Complete.

Definition 2.2 (SpecMine) *Given a specification template represented as a FSA A over an alphabet Σ and an execution trace T over a disjoint alphabet Σ' , i.e., $\Sigma \cap \Sigma' = \emptyset$. The specification mining question (SpecMine) asks whether there exists a total and one-to-one mapping $\rho : \Sigma \rightarrow \Sigma'$ such that the projected trace of T over the range of ρ is accepted by $\rho(A)$, which is the same as A except each transition $\delta(s, a) = t$ is replaced by $\delta(s, \rho(a)) = t$.*

For this to be of interest, it is clear that $|\Sigma| \leq |\Sigma'|$.

The reduction is from the Hamiltonian Path Problem. First let us recall the definition of the problem.

Definition 2.3 (HamPath) *We consider the directed version of the problem: Given a directed graph $G = (V, E)$, does G have a path that visits each vertex $v \in V$ exactly once?*

The problem is well-known to be NP-complete [15]. We will give a polynomial-time reduction from HamPath to SpecMine to show that SpecMine is NP-hard.

Consider an instance of HamPath $G = (V, E)$. We construct a nondeterministic automaton $A = (\Sigma, S, s_0, \delta, F)$ as follows:

- $\Sigma = V$;
- $S = \bigcup_{v \in V} \{s_v\} \cup \{s_0, s^*\}$;
- s_0 is the start state;
- $\forall (u, v) \in E. \delta(s_u, u) = s_v$;
- $\forall u \in V. \delta(s_0, \epsilon) = s_u$;
- $\forall u \in V. \delta(s_u, u) = s^*$;
- $F = \{s^*\}$.

Let $|V| = n$. Let Σ' be an arbitrary alphabet such that Σ' and Σ have the same cardinality (i.e., $|\Sigma'| = |\Sigma|$) and are disjoint (i.e., $\Sigma \cap \Sigma' = \emptyset$). We then construct a trace $T = T_1 \dots T_n$ over Σ' such that $T_i \neq T_j$ for all $1 \leq i \neq j \leq n$.

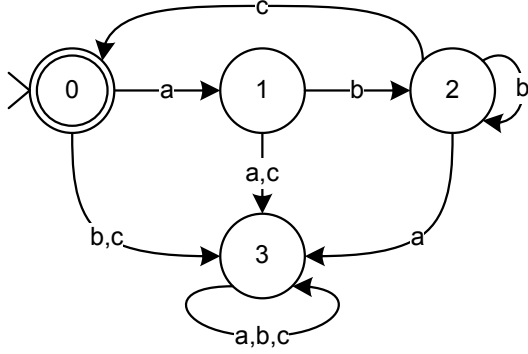


Figure 2: An automaton representing the resource usage pattern.

Lemma 2.4 $G = (V, E)$ has a Hamiltonian path if and only if $\text{SpecMine}(A, T)$.

PROOF.

(\Rightarrow) Assume $G = (V, E)$ has a Hamiltonian path p . We construct a mapping $\rho : \Sigma \rightarrow \Sigma'$ as follows:

$$\rho(p[i]) = T[i]$$

where $p[i]$ denotes the i -th vertex on the path p and $T[i]$ denotes the i -th letter on the trace T ($1 \leq i \leq n$). This mapping shows that T is accepted by A by considering the following path p through A :

$$s_0, s_{p[1]}, s_{p[2]}, \dots, s_{p[n]}, s^*$$

The path p causes A to accept the string

$$\begin{aligned} & \epsilon. \rho(p[1]). \rho(p[2]). \dots. \rho(p[n]) \\ &= \rho(p[1]). \rho(p[2]). \dots. \rho(p[n]) \\ &= T[1]. T[2]. \dots. T[n] \\ &= T \end{aligned}$$

(\Leftarrow) Assume $\text{SpecMine}(A, T)$. Then there exists a mapping $\rho : \Sigma \rightarrow \Sigma'$ such that A has a path p that accepts T . Since ρ is invertible, the path p must have the form:

$$s_0, s_{\psi(T[1])}, \dots, s_{\psi(T[n])}, s^*$$

where ψ denotes ρ^{-1} . Thus $\psi(T[1]), \dots, \psi(T[n])$ is a path of G and it is Hamiltonian. \square

Theorem 2.5 (NP-completeness) SpecMine is NP-complete.

PROOF. We first show that SpecMine is in NP. Its NP-completeness follows because it is clear that the reduction from HamPath to SpecMine is polynomial. SpecMine is in NP because we can generate in polynomial time a mapping ρ and check, also in polynomial time, whether the projected trace is accepted by $\rho(A)$. \square

2.2 The Need for Larger Automata

Consider the automaton in Figure 2. It represents the pattern $(ab^+c)^*$, which describes common programming tasks that involve acquiring a resource, using it, and properly releasing it. It is desirable to mine this specification pattern from programs, but it is either costly (for small examples) or impossible (for moderately sized examples) with current techniques.

In some cases, it is possible to exploit the transitive relationships between smaller properties to form larger properties. In our earlier example, the algorithm discovers the properties $(\text{open, close})^*$, $(\text{open, dispose})^*$, and $(\text{close, dispose})^*$. From these three 2-letter properties and their transitive relationships, we may infer a 3-letter

property: $(\text{open, close, dispose})^*$. The intersection of the expansion of three discovered languages over their combined alphabet is precisely the inferred property. As further motivation for our work, we show that the automaton in Figure 2 has no two-letter decomposition.

Definition 2.6 (Expansion) The expansion of an automaton A with alphabet Σ over an arbitrary alphabet Σ' , $E_{\Sigma'}(A)$, is an automaton, A' , that is equal to A with a looping transition $\delta(q, a) = q$ added to each state q for each letter $a \in \Sigma' \setminus \Sigma$.

The expansion operator is essentially the inverse of projection. For example, the regular expression corresponding to $E_{\{a,b,c\}}((ab)^*)$ is $c^*(ac^*bc^*)^*$. In the context of specification mining, the expansion of a property over the alphabet of all possible trace events yields a language that describes all possible program traces in which the property holds.

Definition 2.7 (Decomposition) A set of automata $\{A'_1, \dots, A'_n\}$ with alphabets $\{\Sigma'_1, \dots, \Sigma'_n\}$ is a decomposition of a finite automaton A with alphabet Σ iff:

$$\bigcup_{i=1}^n \Sigma'_i = \Sigma \quad \bigwedge \quad \bigcap_{i=1}^n \mathcal{L}(E_{\Sigma}(A'_i)) = \mathcal{L}(A)$$

Theorem 2.8 The resource usage automaton in Figure 2, R , represented by the regular expression $(ab^+c)^*$, has no two-letter decomposition.

PROOF. Let Σ be the alphabet of R ($\{a, b, c\}$). We can assume without loss of generality that the decomposition would consist of exactly $\binom{3}{2} = 3$ automata, as any automata over the same subset of the alphabet can be unioned. We refer to these three automata as R_1 , R_2 , and R_3 with respective alphabets $\Sigma_1 = \{a, b\}$, $\Sigma_2 = \{b, c\}$, $\Sigma_3 = \{a, c\}$.

It is clear from the definition of a decomposition that:

$$\mathcal{L}(E_{\Sigma}(R_1)) \cap \mathcal{L}(E_{\Sigma}(R_2)) \cap \mathcal{L}(E_{\Sigma}(R_3)) = \mathcal{L}(R)$$

It then follows that:

$$\begin{aligned} \mathcal{L}(R) &\subseteq \mathcal{L}(E_{\Sigma}(R_1)) \\ \mathcal{L}(R) &\subseteq \mathcal{L}(E_{\Sigma}(R_2)) \\ \mathcal{L}(R) &\subseteq \mathcal{L}(E_{\Sigma}(R_3)) \end{aligned}$$

Applying the projection operator to both sides yields:

$$\begin{aligned} \mathcal{L}(R \mid_{\Sigma_1}) &\subseteq \mathcal{L}(R_1) \\ \mathcal{L}(R \mid_{\Sigma_2}) &\subseteq \mathcal{L}(R_2) \\ \mathcal{L}(R \mid_{\Sigma_3}) &\subseteq \mathcal{L}(R_3) \end{aligned}$$

We then convert these projections into their associated regular expressions.

$$\begin{aligned} (ab^+)^* &\subseteq \mathcal{L}(R_1) \\ (b^+c)^* &\subseteq \mathcal{L}(R_2) \\ (ac)^* &\subseteq \mathcal{L}(R_3) \end{aligned}$$

The language of every two-letter decomposition of R must be included in the intersection of the expansion of these three languages. However, the following trace is accepted by this intersection but *not* by R .

$$a, b, b, c, b, b, a, b, b, c$$

It follows that R has no two-letter decomposition. \square

2.3 Current Approaches

Yang *et al.* describe a basic algorithm, Perracotta, for solving the specification mining problem for input automata with alphabets of size 2. The algorithm creates an n by n matrix that contains the state of every possible assignment of the trace alphabet into the automaton alphabet, where n is the number of unique input symbols. Each (i, j) entry in the matrix is initially set to the starting state of the automaton A .

The algorithm then iterates through the input trace. Each time an input symbol s is seen, its index i is retrieved. It then iterates through each row (corresponding to all $(s, _)$ assignments) and column (corresponding to all $(_, s)$ assignments) and updates the state based on the transition function of A . After each symbol in the trace has been processed, it iterates through the possible assignments and outputs those that are in the final state. The space requirement for this algorithm is clearly $O(n^2)$, and the time complexity is $O(nL)$, where L is the length of the input trace.

Consider a hypothetical extension of this algorithm to three dimensions. In order to track all the assignments, $O(n^3)$ space is now needed. In addition, the scanning of an input symbol s now requires updating all assignments for $(s, _, _)$, $(_, s, _)$, and $(_, _, s)$, of which there are a total of $O(n^2)$. This raises the time complexity to $O(n^2L)$.

In general, this approach uses $O(n^k)$ space and $O(n^{k-1}L)$ time, where k is the size of the alphabet of the template automaton. This is prohibitively costly for alphabets with greater than two letters (Section 4).

3. ALGORITHM DESCRIPTION

In this section, we describe our algorithm in detail. We first give the generic, high-level version (Section 3.1). We then provide some background on binary decision diagrams (Section 3.2), a graph-based representation of Boolean functions that we then use to effectively implement the algorithm (Section 3.3).

3.1 High-level Algorithm

Consider an automaton $(Q, \Sigma, \delta, q_0, F)$ and a trace T over the larger, disjoint trace alphabet Σ' . At a high level, our algorithm constructs a symbolic formula $\varphi : (\Sigma \rightarrow \Sigma') \rightarrow Q$ that represents the current automaton state of each assignment of trace symbols into the pattern alphabet. It then scans each letter in the input trace and symbolically applies the transition function of the automaton to the original formula.

The formula representing the initial configuration maps each possible assignment to the initial automaton state q_0 .

$$\varphi_0 \equiv \left(\bigwedge_{x \in \Sigma} \left(\bigvee_{a \in \Sigma'} x = a \right) \right) \wedge s = q_0$$

We refer to this symbolic formula as a *state configuration*. In our algorithm, we only consider assignments in $\Sigma \rightarrow \Sigma'$ that are one to one; assignments of the same trace letter into two distinct letters of the automaton do not represent interesting specifications.

To apply the transition function δ on a trace letter $a \in \Sigma'$, we update the state of all affected assignments. Consider an arbitrary state configuration φ . We define a transition function, $trans(\varphi, a) = \varphi'$, that applies δ to φ on the input letter a . It is given by the following symbolic formula:

$$trans(\varphi, a) \equiv \bigvee_{(q,x) \rightarrow q' \in \delta} \left((\varphi \wedge \neg(x = a \wedge s = q)) \vee ((\exists s. \varphi \wedge (x = a \wedge s = q)) \wedge s = q') \right)$$

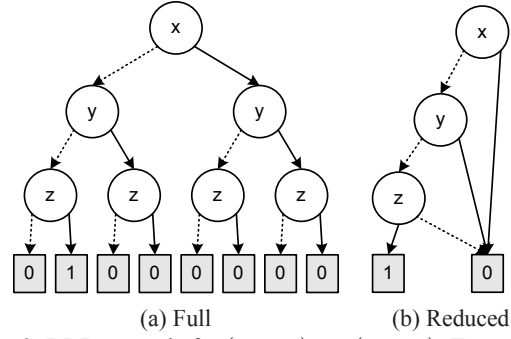


Figure 3: BDD example for $(x \Rightarrow y) \wedge \neg(z \Rightarrow y)$. True edges are solid; false edges are dotted.

The first term of the disjunction, $\varphi \wedge \neg(x = a \wedge s = q)$, captures the state mapping of those assignments that are unaffected by a transition $(q, x) \rightarrow q'$. The second term, $(\exists s. \varphi \wedge (x = a \wedge s = q)) \wedge s = q'$, captures the state mapping of affected assignments. It first projects the affected assignments with an existential quantification and updates their corresponding state to q' .

To implement this algorithm, we now need a suitable representation for the symbolic formulas. We use binary decision diagrams (BDDs) for this because they can compactly represent a large, regular state space. Before presenting our BDD-based algorithm, we first introduce some basic background on BDDs.

3.2 BDD Background

A Binary Decision Diagram (BDD) [6] is a decision tree representation of a Boolean function over a finite set of ordered Boolean variables. Each non-terminal BDD node is associated with a Boolean variable and has exactly two outgoing edges: a **true** edge and a **false** edge. Every BDD has exactly two terminal nodes: a **zero** node and a **one** node.

Given an arbitrary truth assignment into these variables, we can determine if the Boolean function is satisfied by performing a traversal of the decision tree. Starting at the root node, we follow the **true** and **false** edges, depending on our assignment into the associated Boolean variable, until we arrive at a terminal node. If the final node is the **one** node, the function accepts the truth assignment. Similarly, if the final node is the **zero** node, the function rejects the assignment.

BDDs are used in *reduced* form, sometimes referred to as ROBDDs (reduced ordered BDDs). Nodes that represent the same variable and have the same successors can be merged, and nodes with both successor edges pointing to the same node can be removed entirely. ROBDDs are *canonical*: given a specific variable ordering, there is exactly one minimal BDD for a given Boolean formula.

It is because of this that BDDs are often adept at representing extremely large sets, as long as they show a certain degree of regularity. In the programming languages and software engineering fields, BDDs have been used to create scalable pointer analyses [4,22] and represent dynamic traces [26]. Given an efficient variable ordering, the BDD can be sufficiently compressed. However, finding an optimal variable ordering is problem dependent and NP-complete in general [5].

Efficient algorithms exist for performing basic Boolean operations on BDDs, and they run in time proportional to the number of nodes. More importantly, the BDDs are maintained in reduced form throughout the execution of the algorithms.

Consider the Boolean formula $(x \Rightarrow y) \wedge \neg(z \Rightarrow y)$. The decision tree representation of this formula (with variable ordering xyz) appears in Figure 3a.

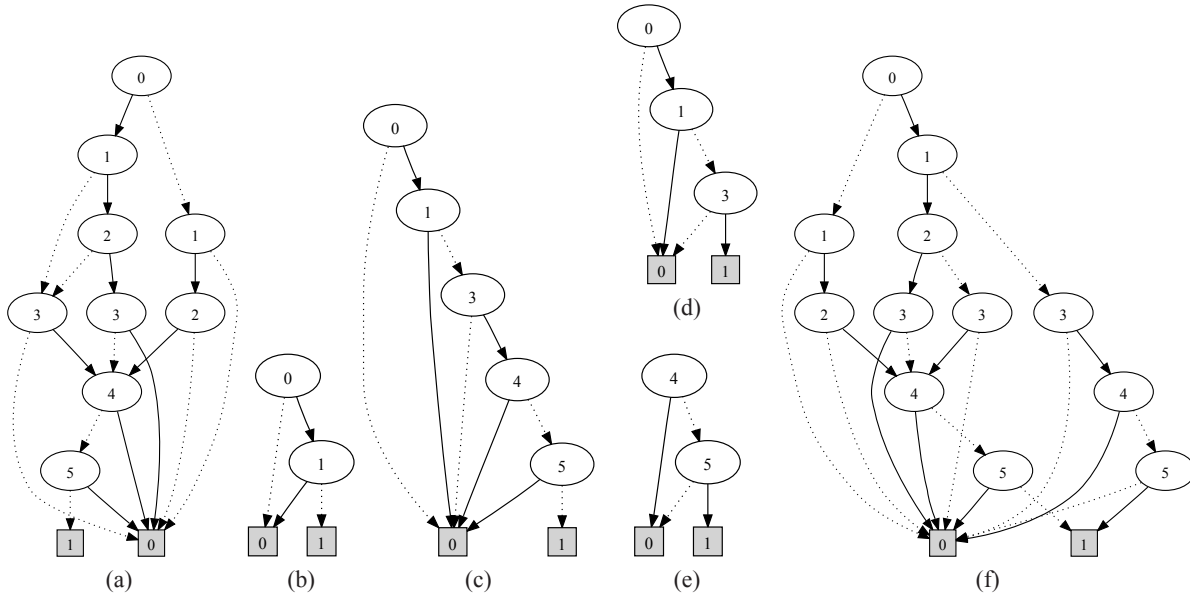


Figure 4: State-tracking BDD example. Nodes are labeled with their corresponding variable numbers. Solid edges are true edges and dotted edges are false edges.

Note that there are several redundant nodes: the graph only requires one terminal node of each type, and several of the z level nodes are identical and can be combined. When combined, several tests become redundant (characterized by identical **true** and **false** edges) and can be eliminated entirely. When all of these transformations have been applied, we are left with the canonical reduced form in Figure 3b.

3.3 BDD-based Algorithm

We implement the general symbolic approach described in Section 3.1 using BDDs. This section describes the implementation of our algorithm in terms of BDD operations.

3.3.1 Running Example

We first give an example to illustrate the core steps of the BDD-based algorithm. Suppose we wish to track the state of six possible assignments of the alphabet $\{a, b, c\}$ into the two-letter, three-state automaton in Figure 1. We can encode each letter and the state in two bits each for a total of six Boolean variables. Using 01 for **a**, 10 for **b**, and 11 for **c**, Figure 4a depicts a BDD representing the assignments in their initial state: $\{(a,b,0), (b,a,0), (a,c,0), (c,a,0), (b,c,0), (c,b,0)\}$.

Suppose we wish to update all assignments that contain an **a** in the first position ($a_ _$) to state 2, *i.e.*, we apply the transition¹ $\delta(a, *) = 2$. We can perform the update on our set as a series of Boolean operations. Figure 4b is the BDD representing all assignments with an **a** in the first position. Note the omission of the other variables: this indicates a “don’t care” assignment. To extract the set of affected assignments, we intersect this BDD with our state BDD, yielding the BDD in Figure 4c.

To remove this extracted set from our main set, we intersect the *complement* of the BDD in 4b with the state. The complement operation involves trading the inbound edges on the terminal **zero** and **one** nodes.

We now wish to update the state on our affected set. We first remove the current state assignment in Figure 4c through *existential*

¹To simplify the example, we are not considering the current state.

quantification over the state variables (Figure 4d). We then intersect the BDD corresponding to the new state, 2 (Figure 4e), and union this updated set into our main set. The final result, which represents the set $\{(a,b,2), (b,a,0), (a,c,2), (c,a,0), (b,c,0), (c,b,0)\}$, is shown in Figure 4f.

3.3.2 Basic Algorithm

Our BDD-based algorithm appears as Algorithm 1. The inputs to our algorithm are 1) a finite trace of symbols, 2) a total and one-to-one mapping of the trace’s unique symbols to an index, and 3) an automaton pattern to mine.

On line 5, we iterate through the alphabet of the input automaton² and allocate enough Boolean variables to encode the unique symbols of the trace (line 6). On line 7, we build up the Cartesian product of all possible assignments.

The function `bddOf(vars, values)` returns a BDD corresponding to the encoding of *values* on variables *vars*. All other Boolean variables are considered to be in a “don’t care” state. On line 10, we place every assignment in our main set in the start state of the automaton.

In our implemented version, we add one additional step. Because assignments with duplicate symbols are not interesting, we build up BDDs corresponding to the $\binom{l-1}{2}$ possibly equal variable sets, *i.e.*, the BDDs in which the two variable sets are equal. We intersect the negation of each of these with our main set.

We then iterate through the trace and repeatedly apply the transition function. Line 18 builds a mask consisting of 1) the input symbol in the appropriate alphabet position and 2) the appropriate current state. We intersect this mask with the global set to retrieve the set of all assignments that are affected by this transition. We union this set with the set “old” for later removal.

We then delete the old state through existential quantification (line 22) and intersect in the new state (line 23). We union this updated set with “new” for later addition.

²For simplicity, we treat the states and alphabet of the automaton as zero-based integers.

Algorithm 1 Pattern-based Specification Mining

```
1: function SPEC_MINE( $T$ :Trace,  $\mu$ :Symbol  $\rightarrow m$ ,  $A$ :( $Q, \Sigma, \delta, q_0, F$ )
2:    $n \leftarrow |\mu|$ 
3:   asgnVars  $\leftarrow \square$ 
4:   set  $\leftarrow \mathbf{True}$ 
5:   for all  $i$  in  $[0..(|\Sigma| - 1)]$  do
6:     asgnVars[ $i$ ]  $\leftarrow \mathbf{bVars}(n)$ 
7:     set  $\leftarrow \text{set} \cap \mathbf{bddOf}(\text{asgnVars}[i], 0..(n - 1))$ 
8:   end for
9:   stateVars  $\leftarrow \mathbf{bVars}(|Q|)$ 
10:  set  $\leftarrow \text{set} \cap \mathbf{bddOf}(\text{stateVars}, q_0)$ 
11:
12:  for all symbol  $t$  in  $T$  do
13:     $i \leftarrow \mu(t)$ 
14:    old  $\leftarrow \emptyset$ 
15:    new  $\leftarrow \emptyset$ 
16:
17:    for all transition  $\delta(q, a) = q'$  do
18:      affected  $\leftarrow \mathbf{bddOf}(\text{asgnVars}[a], i) \cap \mathbf{bddOf}(\text{stateVars}, q)$ 
19:      affected  $\leftarrow \text{affected} \cap \text{set}$ 
20:      old  $\leftarrow \text{old} \cup \text{affected}$ 
21:
22:      affected  $\leftarrow \exists \text{stateVars} . \text{affected}$ 
23:      affected  $\leftarrow \text{affected} \cap \mathbf{bddOf}(\text{stateVars}, q')$ 
24:      new  $\leftarrow \text{new} \cup \text{affected}$ 
25:    end for
26:
27:    set  $\leftarrow \text{set} \cap \overline{\text{old}}$ 
28:    set  $\leftarrow \text{set} \cup \text{new}$ 
29:  end for
30:  return set
31: end function
```

After all relevant transitions have been applied, we subtract the previous assignments (line 27) from the main set and add the updated assignments (line 28).

After execution of the algorithm, the main set contains the state of every possible assignment at the end of the trace. At this point, we intersect this set with the BDD corresponding to the final state and iterate the satisfying assignments to retrieve the potential specifications.

The running state of this algorithm exhibits a high degree of regularity: the algorithm consists of shuffling the numerous assignments into comparatively few partitions defined by the states of the automaton. In practice, BDDs proved to be an appropriate choice for implementation (Section 4).

3.3.3 Practical Improvements

Algorithm 1 precisely and effectively solves the pattern-based specification mining problem as defined in Section 2.1. However, there are several practical issues that must be addressed to make the results more applicable to real systems.

Problem Size Although the use of BDDs greatly improves the tractability of this problem, some traces may still have too many unique events to consider all possible assignments. In our evaluation, the largest trace has 7000 unique events, and our precise algorithm completed its scan in a reasonable amount of time. However, it is conceivable that there exist problems that are too large to solve with this approach.

To reduce the state space, we can partition the original problem into a set of smaller problems. Even a very coarse, conservative partitioning can significantly improve execution speed, and in practice, partitioning is especially effective when using BDDs. Because we process the partitions simultaneously, the BDDs can cache and share nodes between the otherwise unrelated state sets.

Large Solution Sets and Imperfect Traces In general, mining specifications through pattern matching produces a large result set. Previous work [10, 18, 25] on mining *alternating* specifications has largely focused on developing efficient ranking and selection mechanisms. Weimer and Necula [21] focus their search on exceptional control flow paths, and they experiment with several ranking statistics. Yang *et al.* [25] develop a suite of heuristic filters that greatly reduce and refine the solution set.

Yang *et al.* also tailor their analysis to handle potentially *imperfect* traces. Imperfection in traces can originate in flaws in the trace generation mechanism, the lack of context information like threading or object identity, or simply from bugs. To address this, their tool ranks candidate specifications based on the ratio of the number of times a specific assignment passes the final state to the number of times it passes the error state (a failed assignment is reset to the start state on error). This allows the discovery of frequently occurring specifications from potentially buggy or otherwise imperfect traces.

Unfortunately, we cannot directly apply this technique to our analysis. Storing a raw satisfaction count or floating point ratio in the BDD is prohibitively expensive: the set quickly becomes heterogeneous and the memory usage and computation time escalate, causing the computation to diverge.

We address both of these issues with extensions to our basic algorithm. Algorithm 2 differs from Algorithm 1 in that it allows the input traces to contain a certain amount of imprecision and it only returns specifications that occur frequently.

We add an additional set of Boolean variables to our global state. These variables encode a *satisfaction count* for each assignment that can rise and fall as the trace is processed. The declaration of these variables occurs on line 11 of the modified algorithm.

The count is initialized to a specified value that is greater than zero (line 14). If it reaches zero, the specification is dropped (line 35). If it reaches our threshold, we declare it to be satisfied and set it aside (line 24). We no longer execute the transition function on specifications that have either fully failed or have been fully accepted.

Each time an assignment enters an error state (line 33), we decrement this count. Each time it enters a final state, we increment the count nondeterministically with a configurable probability. In both cases, we reset the assignment's state to the start state of the automaton so that execution can continue.

We found that using a threshold of 7 (3 bits), a starting point of 2, and a probability of 0.5 worked well. The nondeterminism at line 29 effectively “amplifies” the count: with a probability of 0.5, we can expect that an accepted specification reaches a final state twice as many times as it would have if the probability was set to 1.

More importantly, we are able to enforce a stricter requirement—a larger satisfaction threshold—without adding additional Boolean variables. We experimented with deterministic thresholds of 16, 32, and 64, and found that even the first extra bit affected the scaling of this approach significantly. The nondeterminism allows us to locate more frequently occurring specifications without incurring this overhead.

Note that we have abstracted away the increment and decrement operators on lines 30 and 34, respectively. Performing arithmetic operations on encoded values in a BDD is non-trivial. To increment the count, we actually need a second set of Boolean variables as temporary storage. We build an “adder” BDD across the two sets that consists of the relation $(n, n+1)$ ($(n, n-1)$ for subtraction) for all n in $0..\text{THRESHOLD} - 1$. Adding then becomes a sequence of Boolean operations: we intersect the value to add with the “adder” BDD and remove the original value by existential quantification.

Algorithm 2 Approximate Pattern-based Specification Mining

```
1: function SPECMINE( $T$ :Trace,  $\mu$ :Symbol  $\rightarrow m$ ,  $A$ :( $Q, \Sigma, \delta, q_0, F$ ))
2:    $n \leftarrow |\mu|$ 
3:    $\text{asgnVars} \leftarrow []$ 
4:    $\text{set} \leftarrow \text{True}$ 
5:   for all  $i$  in  $[0..(|\Sigma| - 1)]$  do
6:      $\text{asgnVars}[i] \leftarrow \text{bVars}(n)$ 
7:      $\text{set} \leftarrow \text{set} \cap \text{bddOf}(\text{asgnVars}[i], 0..(n - 1))$ 
8:   end for
9:    $\text{stateVars} \leftarrow \text{bVars}(|Q|)$ 
10:   $\text{set} \leftarrow \text{set} \cap \text{bddOf}(\text{stateVars}, q_0)$ 
11:   $\text{countVars} \leftarrow \text{bVars}(\text{THRESHOLD} + 1)$ 
12:   $\text{NOTFAILED} \leftarrow \text{bddOf}(\text{countVars}, 1.. \text{THRESHOLD})$ 
13:   $\text{NOTSATISFIED} \leftarrow \text{bddOf}(\text{countVars}, 0..(\text{THRESHOLD} - 1))$ 
14:   $\text{set} \leftarrow \text{set} \cap \text{bddOf}(\text{countVars}, \text{STARTVAL})$ 
15:
16:  for all symbol  $t$  in  $T$  do
17:     $i \leftarrow \mu(t)$ 
18:     $\text{old} \leftarrow \emptyset$ 
19:     $\text{new} \leftarrow \emptyset$ 
20:
21:    for all transition  $\delta(q, a) = q'$  do
22:       $\text{affected} \leftarrow \text{bddOf}(\text{asgnVars}[a], i) \cap \text{bddOf}(\text{stateVars}, q)$ 
23:       $\text{affected} \leftarrow \text{affected} \cap \text{set}$ 
24:       $\text{affected} \leftarrow \text{affected} \cap \text{NOTSATISFIED}$ 
25:       $\text{old} \leftarrow \text{old} \cup \text{affected}$ 
26:
27:       $\text{affected} \leftarrow \exists \text{stateVars} . \text{affected}$ 
28:      if  $q' \in F$  then
29:        if  $\text{nondet}(\text{ADVPROB})$  then
30:           $\text{increment}(\text{affected}, \text{countVars})$ 
31:        end if
32:         $\text{affected} \leftarrow \text{affected} \cap \text{bddOf}(\text{stateVars}, q_0)$ 
33:      else if  $\text{isError}(q')$  then
34:         $\text{decrement}(\text{affected}, \text{countVars})$ 
35:         $\text{affected} \leftarrow \text{affected} \cap \text{NOTFAILED}$ 
36:         $\text{affected} \leftarrow \text{affected} \cap \text{bddOf}(\text{stateVars}, q_0)$ 
37:      else
38:         $\text{affected} \leftarrow \text{affected} \cap \text{bddOf}(\text{stateVars}, q')$ 
39:      end if
40:       $\text{new} \leftarrow \text{new} \cup \text{affected}$ 
41:    end for
42:
43:     $\text{set} \leftarrow \text{set} \cap \overline{\text{old}}$ 
44:     $\text{set} \leftarrow \text{set} \cup \text{new}$ 
45:  end for
46:  return  $\text{set} \cap \text{bddOf}(\text{countVars}, \text{THRESHOLD})$ 
47: end function
```

We then substitute the new value (now in the temporary values) for the original with the BDD `replace` operation.

In practice, this modification scales well. We use this variant of the technique to locate interesting specifications and find bugs.

4. IMPLEMENTATION AND RESULTS

We implemented both versions of our algorithm as part of a practical specification mining tool. This section describes our implementation and presents the results of our empirical evaluation.

4.1 Implementation and Experimental Setup

Our specification miner is a Java application that uses the JavaBDD³ library to provide BDD functionality. The JavaBDD library is a common Java-based abstraction layer for several common BDD packages. We chose to configure the library to use the well-tested BuDDy library.⁴

³<http://javabdd.sourceforge.net/>

⁴<http://www.itu.dk/research/buddy>

We implemented the precise symbolic algorithm (Algorithm 1), the approximate symbolic algorithm (Algorithm 2), and the explicit state tracking algorithm (Section 2.3) for comparison. As described in Section 3.3.3, we also implemented variants that partition the data for scalability.

Our tool is wrapped in a front end that prescans the trace and identifies the unique symbols. It then uses a hash function to build a mapping on to the natural numbers. For convenience, this front end tracks the average execution time of a single element of the trace and provides a progress meter and regular estimates of the time to completion.

Other BDD-based algorithms [4, 22] can show great variability in performance depending on the ordering of the Boolean variables within the BDDs. Compared to the sets represented in these algorithms, our sets are far more saturated and homogeneous: we build the Cartesian product of every possible value. We found that a simple interleaving of all assignment variables yielded the best general performance.

If the shape of the main set changes drastically at some point in the execution of the algorithm, we allow BuDDy's dynamic variable reordering to execute and shuffle the assignment variables using a built in sliding window algorithm. This reordering is triggered with a pathological operation causes a dramatic increase in the number of BDD nodes. In practice, we find that this occurs relatively infrequently.

We collected traces for two large Java projects, Hibernate⁵ and Apache Ant⁶. Each trace consists of a sequence of logged method invocations with their fully qualified class names. We built a trace collection tool using Java bytecode instrumentation provided by the JRat⁷ toolkit. Our tool takes as input a Jar (Java archive) or directory and recursively instruments all method calls with hooks into our logging code. We also slipstream our logging classes into the instrumented Jar files to avoid the task of changing the projects' runtime profiles.

Our traces are generated from both normal usage of each project and extensive testing using each project's respective system test suite.

4.2 Results for the Precise Algorithm

We first tested the scalability of our precise algorithm on four small to medium sized traces. For comparison, we also tested the basic algorithm (Section 2.3) that maintains the state of every assignment explicitly in a matrix. Figure 5a contains the results of mining the alternating pattern found in Figure 1. On these smaller traces, the basic algorithm outperforms our symbolic algorithm. However, the gap narrows as the number of unique events increase.

Figure 5b shows the results of mining the 3-letter resource usage pattern found in Figure 2. When the extra dimension is added, the symbolic algorithm is orders of magnitude more scalable. The basic algorithm quickly diverges in terms of memory usage and is three orders of magnitude slower.

We then tested the algorithms on our two largest traces. Figure 6 contains the results of these tests. When mining the 2-letter pattern, the symbolic algorithm exploits the regularity of the solution set and both executes faster and in less space than the basic algorithm.

We were naturally unable to mine the 3-letter pattern using the basic algorithm, as this would involve individually tracking the states of over 50 billion and 300 billion potential assignments for the two traces, respectively. The symbolic algorithm was able to

⁵An object-relational mapping tool.

⁶A make-like build tool for Java.

⁷<http://jrat.sourceforge.net>

Trace		Basic Algorithm		Symbolic		Basic Algorithm		Symbolic	
Uniq. Evt.	Length	Time	Max Mem.	Time	Max Mem.	Time	Max Mem.	Time	Max Mem.
338	24,470	0.17s	18 MB	0.84s	52 MB	3m 31s	196 MB	10.4s	54 MB
591	94,789	0.79s	29 MB	2.7s	52 MB	42m 3s	942 MB	34.5s	54 MB
664	227,595	3.0s	30 MB	5.6s	64 MB	2h 8m	1.2 GB	68.4s	64 MB
981	1,435,613	18.4s	37 MB	28.8s	66 MB	27h 59m	4.0 GB	2m 53s	66 MB

(a) The alternating ($|\Sigma| = 2$) pattern.(b) The resource usage ($|\Sigma| = 3$) pattern.

Figure 5: Results of mining patterns using the precise algorithm on four small traces.

Trace		Basic Algorithm		Symbolic		Symbolic	
Uniq. Evt.	Length	Time	Max Mem.	Time	Max Mem.	Time	Max Mem.
3821	21,055,090	22m	91 MB	7m 30s	68 MB	5h 17m	302 MB
7000	49,779,538	3h 19m	238 MB	23m	119 MB	13h 34m	589 MB

(a) The alternating pattern.

(b) The resource usage pattern.

Figure 6: Execution of the precise algorithm on large traces.

complete this computation in a reasonable amount of time and with low memory usage.

Overall, the symbolic approach greatly extends the tractability of this problem.

4.3 Results for the Approximate Algorithm

Our approximate algorithm yields more practically relevant solution sets. In this section, we evaluate its performance and illustrate previously unknown bugs that we located using specifications derived from this analysis.

We used the same two large traces, which were generated from executions of Apache Ant and Hibernate, respectively. We evaluated both the precise and practical algorithms using a package-level partitioning: we only considered assignments in which the involved methods were defined in classes of the same package. Figure 7a describes the partitioning of the traces. Code is seldom uniformly distributed into packages, so we have included the maximum partition size as well. This value is closely related to scalability: the number of potential assignments is cubic in the size of any given partition.

4.3.1 Performance

Figures 7b and 7c contain the results of executing our symbolic algorithms on these traces using the package-level partitioning. Both algorithms execute in similar amounts of space and time.

Executing the exact algorithm on the Apache Ant trace, however, produces an enormous amount of potential specifications. This is likely because Ant’s overall build process emulates the resource usage pattern at a higher level: builds are characterized by several sub-builds surrounded by a start and end. This causes the recognition of virtually every build process method as a “usage” method between the build start and end methods.

The use of the approximate algorithm substantially prunes this result set. Although we potentially admit more specifications by allowing failed specifications additional chances to be accepted, we restrict the set by admitting only frequently occurring specifications.

4.3.2 Result Quality

For the Hibernate project, we inferred several properties of the `Session` object of the form:

```
a: sessionFactoryImpl.openSession(Connection,...)
b: AbstractSessionImpl.createQuery(String)
c: AbstractSessionImpl.setClosed()
```

```
1 public static void copyResource(Resource source, Resource dest,
2   String inputEncoding, String outputEncoding,
3   Project project) {
4   BufferedReader in = null;
5   try {
6     InputStreamReader isr = null;
7     if (inputEncoding == null) {
8       isr = new InputStreamReader(source.getInputStream());
9     } else {
10      isr = new InputStreamReader(source.getInputStream(),
11        inputEncoding);
12    }
13    in = new BufferedReader(isr);
14    // getToken() indirectly calls the read() method
15    String line = lineTokenizer.getToken(in);
16    while (line != null) {
17      line = lineTokenizer.getToken(in);
18    }
19  } finally {
20    FileUtils.close(in); // null-safe close()
21  }
22 }
```

Figure 8: A previously unknown bug in Apache Ant 1.7.0. Only relevant lines are shown.

where the a) and c) assignments involved all combinations of the various signatures of `openSession` and the various `close/setClosed` methods, and the b) assignment consisted of frequently executed methods of the `Session` class. These specifications are documented and have been targets for previous automated specification tools [21].

Most spurious specifications were either redundant or artifacts of control flow: we use flat traces, so the execution of one method within the body of another appears as subsequent invocations. To mitigate this issue, we can use a filtering step, like that of Yang *et al.* [25], that prunes assignments with a direct control flow relationship.

The Ant build system interacts with many types of resources that fit our pattern. These include various archive formats, flat files, and resources defined by URLs. We inferred several properties of the form:

```
a: ZipResource.getInputStream()
b: InflaterInputStream.read()
c: InflaterInputStream.close()
```

We inspected our results and the source code, and we discovered that each of these resources inherits from a common class, `Resource`. Analyzing the usages of this class and specifically searching for resource allocation bugs, we discovered the flaw in Figure 8.

Trace Data				Exact			Approximate		
Source	Uniq. Evts.	Partitions	Max Part. Size	Time	Max Mem.	Sat. Count	Time	Max Mem.	Sat. Count
Ant	3821	32	1164	21m	90 MB	1,383,532	27m	122 MB	7,724
Hibernate	7000	68	703	41m	117 MB	55,374	48m	146 MB	13,608

(a) Partitioning. (b) Exact, resource usage pattern. (c) Approx, resource usage pattern

Figure 7: Results of running the precise and approximate versions on partitioned versions of the larger traces.

The parameter `inputEncoding` is an unsanitized user input string. Note that the allocation and release of the resource are properly enclosed in a try/finally block. However, line 10 contains a problem: the creation of an `InputStreamReader` with a user-supplied character set string can fail, while the parameterless creation of `Readers` on lines 8 and 12 cannot. If the allocation on line 10 does fail, the call to `getInputStream` will be eagerly evaluated and the resource will be allocated. The variable `in` will never be assigned, and the null-safe `close` call on line 20 will not close the resource, causing a leak.

Both of the listed patterns are alike in the sense that all participants are not necessarily of the same type. This is a key advantage of the pattern-based approach in general: the scope of the analysis is not limited to a small set of suspected methods that must be decided by a user.

An interesting example that illustrates this advantage occurs in the Ant project:

```
a: MailMessage(String,int)
b: MailPrintStream.write(byte[],int,int)
c: MailMessage.sendAndClose()
```

A programmer utilizing the public interface of this class might be confused⁸ by the method naming and assume that all socket connections are performed in the final `sendAndClose` method.

The frequency of the discovered pattern suggests that the methods of this class actually follow an allocation pattern. On inspection of the source code, we noted that the first constructor call does in fact allocate and connect a socket, the second `MailPrintStream` object uses it, and the final `sendAndClose` closes it.

Figure 9 illustrates a usage of this class in the Ant source code. The proper allocation and use of the `MailMessage` resource should be enclosed in a try/finally block, but all exceptions are thrown to the caller. This could potentially cause resource leaks.

While evaluating the solution sets produced by our algorithms, we noted several instances where transitive relationships might be used to chain smaller specifications into larger ones. As discussed in Section 2.2, previous work has shown this to be useful for alternating properties. Our work enables other new fundamental building blocks, including three-letter patterns with loops. We leave for future work a study of these constructed properties.

5. RELATED WORK

The literature on specification mining is varied, but many tools and techniques share similar characterizations of properties: finite automata that describe correct behavior. Ammons *et al.* [2] develop a specification miner, Strauss, that mines specification by learning a probabilistic finite state automaton. Unlike our approach, Strauss requires the input alphabet of the automaton to be specified, but it does have the potential to find more complex specifications.

Whaley *et al.* [23] present an alternative method: the user specifies the input alphabet of the automaton (a Java API), and the al-

```
1 private void sendMail(String mailhost, int port, String from, String
   replyToList, String toList, String subject, String message)
   throws IOException {
2   MailMessage mailMessage = new MailMessage(mailhost, port);
3   mailMessage.setHeader("Date", DateUtils.getDateForHeader());
4
5   mailMessage.from(from);
6   if (!replyToList.equals("")) {
7     StringTokenizer t = new StringTokenizer(replyToList, " , ",
       false);
8     while (t.hasMoreTokens()) {
9       mailMessage.replyTo(t.nextToken());
10    }
11  }
12  StringTokenizer t = new StringTokenizer(toList, " , ", false);
13  while (t.hasMoreTokens()) {
14    mailMessage.to(t.nextToken());
15  }
16
17  mailMessage.setSubject(subject);
18
19  PrintStream ps = mailMessage.getPrintStream();
20  ps.println(message);
21
22  mailMessage.sendAndClose();
23 }
```

Figure 9: Another previously unknown bug in Apache Ant 1.7.0.

gorithm identifies and prunes illegal transitions based on the existence state-checking code that throws exceptions on errors. This technique relies on defensively-programmed components that imply correct usage through error checking.

More recently, Shoham *et al.* [20] present a technique based on abstract interpretation that is capable of learning arbitrarily complex specifications. The abstract value is the automaton itself, and it is constructed as the program is interpreted. This technique is accurate, but like the previously mentioned tools, the alphabet of the automaton must be known before the analysis begins.

Ramanathan *et al.* [18, 19] create a static analysis that uses predicate mining to infer *function precedence protocols*. These rules take the form of “when *b* is called, *a* must have been called at least once.” When viewed as a specification pattern, these rules take the form of the regular expression (a^+b) . Our work is capable of locating more complex properties. In addition, as we show in Section 2.2, it is generally impossible to assemble two letter properties like this to form our more complex properties.

Kremenek *et al.* [16] use *annotation factor graphs* to probabilistically assign annotations to functions and form specifications. This technique allows the user to incorporate other domain-specific information into the analysis, like a belief in the overall ratio of allocators to deallocators, as components in the factor graph. The user must still specify the set of annotations that can be inferred, which is similar to the specification of the potential alphabet of an automaton in the sense that it limits the results of the analysis.

One tool that does not follow this pattern of deriving automata-described temporal properties is PR-Miner [17]. This tool, presented by Li and Zhou, mines *association* properties from programs by correlating related function calls using *frequent itemset mining*.

⁸For fairness, we point out that this proper usage pattern is documented in the code comments. It does not mention the resource allocation issues, though.

This work, like ours, is highly scalable. However unlike our tool, PR-Miner requires no input from the user other than the program (our technique requires a specification template). However, these association properties are not as strict as temporal properties: the members of the properties are only related by frequent association and not by the order or frequency of invocation.

There has been recent work on dynamic analysis as well. Ernst *et al.* [11] have developed Daikon (refined for object oriented systems in [8]) and Hangal and Lam [13] have developed DIDUCE. These tools locate program invariants by monitoring the runtime state of a program and attempting to match invariant templates to expressions. These tools are able to infer simple properties, like $a \neq 0$, but at present do not infer temporal properties. When viewed as an online algorithm, our work can be thought of as a Daikon-like tool for temporal properties: we begin by setting up a large set of potentially valid properties and prune them as execution continues.

6. CONCLUSIONS

In this paper, we have precisely defined the pattern based specification mining problem and shown that its general form is NP-complete. We have provided a novel symbolic algorithm that greatly expands the computational tractability of this problem. Our algorithm exploits the regularity of the running state and solution set of current approaches to form a compact, efficient symbolic approach. We have implemented our algorithm as a practical tool using binary decision diagrams and have used it to find meaningful specifications in real systems. These specifications led us to discover previously unknown bugs in large, real-world systems. For future work, we plan to investigate the synthesis of larger specifications from our inferred properties, and we are also interested in adapting our approach to locate other frequently-occurring specification patterns.

7. REFERENCES

- [1] R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*.
- [2] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*.
- [3] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*.
- [4] M. Berndt, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. In *Proceedings of PLDI*, pages 103–114, 2003.
- [5] B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Trans. Comput.*, 45(9):993–1002, 1996.
- [6] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.
- [7] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*.
- [8] C. Csallner and Y. Smaragdakis. Dynamically discovering likely interface invariants. In *Proceedings of ICSE*, pages 861–864, 2006.
- [9] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*.
- [10] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*.
- [11] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *Proceedings of ICSE*, pages 449–458, 2000.
- [12] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of PLDI*, pages 234–245, 2002.
- [13] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of ICSE*, pages 291–301, 2002.
- [14] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 132–136, 2004.
- [15] R. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*. New York.
- [16] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: inferring the specification within. In *USENIX'06: Proceedings of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation*, pages 12–12, 2006.
- [17] Z. Li and Y. Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of ESEC/FSE-13*, pages 306–315, 2005.
- [18] M. K. Ramanathan, A. Grama, and S. Jagannathan. Path-sensitive inference of function precedence protocols. In *Proceedings of ICSE*, pages 240–250, 2007.
- [19] M. K. Ramanathan, A. Grama, and S. Jagannathan. Static specification inference using predicate mining. In *Proceedings of PLDI*, pages 123–134, 2007.
- [20] S. Shoham, E. Yahav, S. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. In *Proceedings of ISSTA*, pages 174–184, 2007.
- [21] W. Weimer and G. Necula. Mining temporal specifications for error detection. In *Proceedings of TACAS*, pages 461–476, 2005.
- [22] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of PLDI*, pages 131–144, 2004.
- [23] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 218–228, 2002.
- [24] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*.
- [25] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: Mining temporal API rules from imperfect traces. In *Proceedings of ICSE*, pages 282–291, 2006.
- [26] X. Zhang, R. Gupta, and Y. Zhang. Efficient forward computation of dynamic slices using reduced ordered binary decision diagrams. In *Proceedings of ICSE*, 2004.