Mining Temporal Specifications for Error Detection

Westley Weimer and George C. Necula
{weimer,necula}@cs.berkeley.edu

University of California, Berkeley

Abstract. Specifications are necessary in order to find software bugs using program verification tools. This paper presents a novel automatic specification mining algorithm that uses information about error handling to learn temporal safety rules. Our algorithm is based on the observation that programs often make mistakes along exceptional control-flow paths, even when they behave correctly on normal execution paths. We show that this focus improves the effectiveness of the miner for discovering specifications beneficial for bug finding.

We present quantitative results comparing our technique to four existing miners. We highlight assumptions made by various miners that are not always born out in practice. Additionally, we apply our algorithm to existing Java programs and analyze its ability to learn specifications that find bugs in those programs. In our experiments, we find filtering candidate specifications to be more important than ranking them. We find 430 bugs in 1 million lines of code. Notably, we find 250 more bugs using per-program specifications learned by our algorithm than with generic specifications that apply to all programs.

1 Introduction

Software remains buggy and testing is still the dominant approach for detecting software errors. The difficulties and costs of testing have helped to push forward techniques that automatically find classes of errors statically [4–8, 10, 12, 16] or dynamically [13–15, 17]. Such program verification tools can point out bugs or provide guarantees about the absence of some mistakes.

Invariably, however, verification tools require *specifications* that describe some aspect of program correctness. Creating correct specifications is difficult, time-consuming and error-prone. Verification tools can only point out disagreements between the program and the specification. Even assuming a sound and complete tool, an imperfect specification can still yield false positives by pointing out non-bugs as bugs or false negatives by failing to point out desired bugs. Crafting specifications typically requires program-specific knowledge.

One way to reduce the cost of writing specifications is to use implicit language-based specifications (e.g., null pointers should not be dereferenced) or to reuse standard library specifications (e.g., [4, 16]). More recently, however,

a variety of attempts have been made to infer program-specific temporal specifications and API usage rules [1, 2, 8, 17] automatically. These *specification mining* techniques take programs (and possibly dynamic traces, or other hints) as input and produce candidate specifications as output. In general, specifications could also be used for documenting, refactoring, testing, debugging, maintaining, and optimizing a program. We focus here on finding and evaluating specifications in a particular context: given a program and a generic verification tool, what specification mining technique should be used to find bugs in the program and thereby improve software quality? Thus we are concerned both with the number of "real" and "false positive" specifications produced by the miner and with the number of "real" and "false positive" bugs found using those "real" specifications.

We propose a novel technique for temporal specification mining that uses information about program error handling. Our miner assumes that programs will generally adhere to specifications along normal execution paths, but that programs will likely violate specifications in the presence of some run-time errors or exceptional situations. Intuitively, error-handling code may not be tested as often or the programmer may be unaware of sources of run-time errors. Taking advantage of this information is more important than ranking candidate policies.

The contributions of this paper are as follows:

- We propose a novel specification mining technique based on the observation that programmers often make mistakes in exceptional circumstances or along uncommon code paths.
- We present a qualitative comparison of five miners and show how some miner assumptions are not well-supported in practice.
- Finally, we give a quantitative comparison of our technique's bug-finding powers to generic "library" policies. For our domain of interest, mining finds 250 more bugs. We also show the relative unimportance of ranking candidate policies. In all, we find 69 specifications that lead to the discovery over 430 bugs in 1 million lines of code.

In Section 2 we describe temporal safety specifications. We present our specification mining algorithm in Section 3. In Section 4 we describe some existing specification mining algorithms, leading up to a qualitative comparison of various techniques in Section 5. We describe our experience running our miner on 1 million lines of code in Section 6, comparing its bug-finding powers to another technique and to generic "library" specifications.

2 Temporal Safety Specifications

In general, a specification mining technique takes a program as input and produces one or more candidate specifications with respect to a set of interesting program events. The program is typically presented to the miner in the form of a set of static or dynamic *traces*, each of which is a sequence of events and annotations (e.g., data values, records of raised exceptions). Static traces are

```
Session sess = sfac.openSession();
Transaction tx;
try {
    tx = sess.beginTransaction();
    // do some work
    tx.commit();
} catch (Exception e) {
    if (tx != null) tx.rollback();
    throw e;
} finally
    sess.close();
```

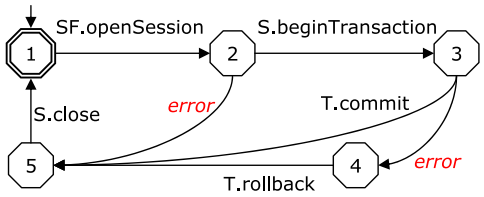

Fig. 1. `Session` temporal safety policy from `hibernate2` **class documentation pseudocode** along with a formulation as a finite state machine over a six-event alphabet. Edge label events are either successful method invocations or method *error*s. Other transitions involving these six events violate the policy, but other events (e.g., `S.find`) are not constrained.

generated from the program source code. Dynamic traces are produced by running the program against a workload. In practice, *events* are usually taken to be context-free function calls.

Mined *specifications* (or *policies*) are typically finite state machines with events as edges. A run of the program adheres to the policy if it generates a sequence of events accepted by the FSM. Such policies commonly limit how an interface may be invoked (e.g., `close` cannot be called before `open` and must be called after it). Many program verifiers can check such FSM properties, either per-object (as a form of typestate) or globally. Ammons et al. [2] present a more formal treatment of the mining problem.

As a concrete example, we consider a policy for the interfaces of the `SessionFactory`, `Session` and `Transaction` classes in the `hibernate2` program, a 57k LOC framework that provides persistent Java objects [11]. The `Session` class is the central interface between `hibernate2` and a client. The `Session` documentation includes explicit pseudocode and an injunction that clients should adhere to it. The code and five-state FSM specification are shown in Figure 1. We denote `SessionFactory` by `SF`, `Session` by `S`, and `Transaction` by `T`. A typical use of this interface would visit states 1 through 3, "do some work" there (involving events like `S.flush` and `S.save` that are not part of the input alphabet of the FSM and thus do not affect it), and then visit 5 and return to 1. In the next section we discuss our mining algorithm using this specification as a concrete example.

3 Specification Mining Algorithm

Our work on specification mining was motivated by observations of run-time error handling mistakes. Based on previous work examining such mistakes [16] we believe that client code frequently violates simple API specifications in exceptional situations (i.e., in the presence of run-time errors). We found such

bugs using generic "library" specifications (e.g., `Socket` and `File` open/close rules), but we believed that we would be able to have a greater impact on software quality by looking for program-specific mistakes. Our mining algorithm produces policies dealing with resource leaks or forgotten obligations. We have found that programs repeatedly violate such policies, especially when run-time errors are involved. Our technique is in the same family as that of Engler et al. [8] but is based on assumptions about run-time errors, chooses candidate event pairs differently, presents significantly fewer candidate specifications and ranks presented candidates differently.

We attempt to learn pairs of events $\langle a, b \rangle$ corresponding to the two-state FSM policy given by the regular expression $(ab)*$. For example, from traces generated by the state machine in Figure 1 we might learn $\langle$`SF.openSession`, `S.close`$\rangle$, because every accepting sequence that transitions from state 1 to state 2 via `SF.openSession` must also transition from state 5 to state 1 via `S.close`. We learn multiple candidate specifications per program and present a ranked list to the user. For example, we might learn the candidate specification $\langle$`SF.openSession`, `T.rollback`$\rangle$. Unlike some mining algorithms that produce detailed policies that must be manually debugged or modified, we produce simple policies that are designed to be accepted or rejected. With this approach we will not be able to learn the "complete" policy in Figure 1. However, the full policy is closely approximated by $\langle$`SF.openSession`, `S.close`$\rangle$ and $\langle$`S.beginTransaction`, `T.commit`$\rangle$.

In a normal execution, events a and b may be separated by other events and difficult to discern as a pair. After an error has occurred, however, the cleanup code is usually much less cluttered and contains only operations required for correctness. Intuitively, a programmer who is aware of the specification will have included b in an exception handler, `finally` block, or other piece of cleanup code, making it easier to pick up than in a normal execution path. The pseudocode in Figure 1 demonstrates this sort of cleanup for the `T.rollback` and `S.close` events. If `S.close` is the only legal way to discharge a `Session` obligation, we expect to see `S.close` in well-written cleanup code.

We classify intra-procedural static traces as "error" traces if they involve exceptional control flow. These are the traces containing at least one method call that terminates with raising of an exception. Such exceptions are assumed to signal run-time errors or unusual situations. Traces in which no such exceptions are raised are "normal" traces. In Figure 1, a normal trace of events would involve the state sequence 1–2–3–5–1. An error trace would visit 1–2–5–1 or 1–2–3–4–5–1.

3.1 Filtering Candidate Specifications

Let E_{ab} be the number of error traces that have a followed by b, and let E_a be the number of error traces that have a but not b. We define N_{ab} and N_a similarly for normal traces and use them for ranking later. Given a set of traces, we consider all event pairs $\langle a, b \rangle$ from those traces such that all of the following occur:

Exceptional control flow (ex). Our novel filtering criterion is that event b must occur at least once in some cleanup code (e.g., a `catch` or `finally` block): we require $E_{ab} > 0$. We assume that if the policy is important to the programmer, language-level error handling will be used at least once to enforce it. In `hibernate2`, the `SF.openSession` and `S.beginTransactions` events never occur in cleanup code, thus ruling them out as the second event in a pair. The `T.commit`, `T.rollback` and `S.close` events all do occur in cleanup code, however. Other miners limit events to those on a user-specified list.[1] We prefer to automate the creation of this list because of the cost of acquiring specific knowledge about each target program. However, if such domain knowledge is available, it can be used instead of, or in addition to, the default from cleanup code. The occurrence of the event in normal execution traces will be used in Section 3.2 to rank candidate specifications.

One error (oe). There must at least one *error* trace with a but without b: we require $E_a > 0$. We are here only interested in learning specifications that will lead to finding program errors, and we assume that the programmer will make mistakes in the handling of exceptional situations.

Same package (sp). Events a and b must be declared in the same package. For example, we assume that no temporal specification will be concerned with the relative order of an invocation of an `org.apache.xpath.Arg` method and a `net.sf.Hibernate.Session` method from separate libraries. The user can specify wider or narrower related groups if such information is available.

Dataflow (df). Every value and receiver object expression in b must also be in a. When dealing with static traces we require that every non-primitive type in b also occur in a. We thus assume that `Session SessionFactory.openSession()` may be followed by `void Session.close()` but forbid the opposite ordering. Intuitively, this also corresponds to finding edges that share the same node in policies like Figure 1. This notion is in contrast to other miners where a more precise dataflow analysis rules out some unwanted specifications. In our experiments this lightweight dataflow requirement has been sufficient to capture our intuitive notion of correlated events.

3.2 Ranking Candidate Specifications

In order to improve the usability of this technique, we present to the user a ranked list of the candidate specifications that satisfy the criteria described above. Our heuristics will assign higher ranks to candidates that are more likely to be real policies. We do not rank policies based on the number of bugs the policy would find in the program. However, as we will see in Section 6, ranking plays a much smaller role than eliminating extraneous candidates.

We assume $\langle a, b \rangle$ is more likely to be a policy if the programmer intends to adhere to it many times. We assume that normal traces represent the intent of the programmer and that some error traces represent unforeseen circumstances

[1] For example, in Engler et al. [8] the list includes functions whose names contain the substrings "lock", "unlock", "acquire", etc.

Event a	Event b	Real	N_a	N_{ab}	E_a	E_{ab}	Filters	rank	z-rank	z-rank$\|_N$
`SF.openSessi`	`S.close`	Yes	3	100	1348	1040	ex oe sp df	0.971	-73.5	2.40
`S.beginTrans`	`S.close`	?	2	56	1037	501	ex oe sp df	0.966	-73.3	1.66
`S.beginTrans`	`T.commit`	Yes	2	56	565	973	ex oe sp df	0.966	-33.9	1.66
`S.flush`	`S.close`	no	9	39	200	473	ex oe sp df	0.812	-17.0	-2.02
`T.commit`	`S.close`	?	1	57	474	504	ex oe sp	0.983	-38.5	2.10
`S.beginTrans`	`S.save`	no	4	54	37	1501	oe sp df	0.931	9.90	0.788
`SF.openSessi`	`T.commit`	?	47	56	1415	973	ex oe sp	0.544	-81.0	-12.1
`SF.openSessi`	`println`	no	82	21	2121	267	ex oe df	0.204	-130	-23.4

Fig. 2. Static trace observations for `Session` events in `hibernate2`. The "real" column indicates whether $\langle a, b\rangle$ is definitely (Yes), possibly (?) or definitely not (no) a valid policy based on Figure 1. N_a is the number traces with a but not b, N_{ab} is the number of normal traces with a followed by b. E_a and E_{ab} measure the same figures for error traces. The "Filters" column indicates which of our filtering requirements the pair meets. Only the first four pairs meet the requirements and would be reported as candidates by our algorithm. The "rank" column reports $N_{ab}/(N_a + N_{ab})$ and high values indicate more likely specifications. The "z-rank" column shows the z-statistic applied to all traces as in Engler et al. [8], while the "z-rank$|_N$" column shows the z-statistic restricted to normal traces.

likely to contain bugs; thus we rank pairs according to the fraction of *normal traces* containing a in which a is followed by b.

Our ranking for a candidate $\langle a, b\rangle$ is $N_{ab}/(N_{ab} + N_a)$. The best ranking is 1, and a reported specification with rank 1 has a followed by b in all normal paths.

Figure 2 shows observations for `Session`-related events on a set of static traces. All eight pairs could potentially be policies, but our requirements in Section 3.1 filter out the last four. Since `SF.openSession` does not occur in any error-handling code, we do not consider pairs like $\langle$`S.close`, `SF.openSession`$\rangle$. As desired, we rule out pairs like $\langle$`SF.openSession`, `T.commit`$\rangle$ with our dataflow requirement (there is no `Transaction` object available in event a). Our package requirement correctly rules out policies involving `printf`-like logging methods. Finally, while we cannot rule out pairs like $\langle$`S.flush`, `S.close`$\rangle$ (where `S.flush` is one of the "do some work" options that would occur at state 3 of Figure 1), we rank it lower because a smaller fraction of normal paths have that pairing (e.g., in Figure 2 that pair ranks 0.812 while the best pair involving `S.close` ranks 0.971).

The z-rank and z-rank$|_N$ columns of Figure 2 show the result of using the z-statistic for proportions [9], an alternative ranking scheme, to rank candidate specifications, with the z-rank$|_N$ column being computed over normal traces only. The z-rank was used by Engler et al. [8]. The z-statistic increases with the total number of observations involving a and decreases with the number of observations involving a but not b. Ignoring some constant factors, z-rank$|_N$ is equal to our ranking multiplied by $\sqrt{N_a + N_{ab}}$. We provide an empirical comparison of these three rankings in Section 6.

4 Other Specification Mining Techniques

We now describe the main characteristics of several existing mining approaches.

Strauss. Ammons et al. [2] present a miner in which events from dynamic traces that are related by traditional dataflow dependencies form a *scenario*. The user provides a *seed* event and a maximum scenario size N. A scenario contains at most N ancestors and at most N descendants of the seed event. The seed can be any interesting event that is assumed to play a role in the specification. Such scenarios are fed to a probabilistic finite state machine learner. The output of the learner, a single policy, is minimized and may further be "cored" by removing infrequently traversed edges or "debugged" and simplified with the user's help [3].

WML-static. Whaley et al. [17] propose two methods for deriving interface specifications for classes based on an explicit representation of typestate in member fields.

In the first, the user specifies a class in the program. Traces are generated statically by considering all pairs $\langle a, b\rangle$ of invocations for methods a and b of that class. If b conditionally raises an exception when a field has a certain constant value and a always sets that field to that value, $\langle a, b\rangle$ is considered a violation of the interface policy. For example, the `close` method might set the field `opened` to false, and the `read` method might raise an exception if `opened` is false. The single final specification consists of all other pairs $\langle a, b\rangle$, represented as an NFA with one state per method. This miner explicitly looks for "a must not be followed by b" requirements, and by considering all possible method pair interactions it discovers what can follow a as well. In our experiments, we used an extended version of the miner that considers multiple fields and inlines boolean methods.

JIST. The JIST tool of Alur et al. [1] refines the WML-static miner by using predicate abstraction for a more precise dataflow analysis. The user specifies a class and an undesired exception, as well as providing a set of predicates and a specification of size k. A boolean model of the class is constructed based on the predicate set, and a model checker determines if invoking a sequence of methods raises the given exception. If it can, that sequence is removed from the specification. The process finds the most permissive policy of that size that is safe with respect to the predicates and the exception. As with Strauss, the output of the analysis is minimized using an off-the-shelf FSM library. In a WML-static policy, states represent the last invoked method. In JIST, states represent predicate valuations, which in turn represent object state. For example, JIST could produce a policy in which the sequence $\langle a, b\rangle$ is allowed but $\langle a, a, b\rangle$ is not. Thus, in JIST's more general policies, states do not correspond directly to the last method invocation.

WML-dynamic. Whaley et al. [17] also present a dynamic trace analysis that learns a *permissive* policy for a given class. Such a permissive specification is the most restrictive policy that accepts all of the training traces. Each field of the class is considered separately. Only events representing client calls to methods of that class that read or write that field are examined. If a is immediately followed

by b in the trace, an edge from a to b is added to the policy. The single output policy for the class is formed from the per-field policies.

ECC. Engler et al. [8] describe a technique for mining rules of the form "b must follow a" as part of a larger work on may-must beliefs, bugs, and deviant behavior. If b follows a in any trace, the event pair $\langle a,\ b\rangle$ is considered as a candidate specification.

A pair $\langle a, b\rangle$ is a candidate policy if the events a and b are related by dataflow and if there are both traces in which a is followed by b and traces in which a is not followed by b. A series of dependency checks is employed: two events are related if they have either the same first argument, or have no arguments, or if the return value from the first passed as the sole argument to the second. The user may also restrict attention to a certain set of methods.

ECC produces a large number of candidate policies. Engler et al. use the z-statistic for proportions to hierarchically rank candidates. The z-statistic measures the difference between the observed ratio and an expected ratio p_0. Engler et al. use the ranking because it grows with the frequency with which the pair is observed together and decreases with the number of counter-examples observed. They take $p_0 = 0.9$ based on the assumption that perfect fits are uninteresting in bug-finding and that error cases are found near counter-examples. In our experiments we have found that ECC's assumptions tend to hold true for normal traces but not for error traces (where the frequency counts are quite high if the traces are static and often quite low if the traces are dynamic).

5 Qualitative Comparison of Mining Techniques

In this section we present experiments comparing these mining techniques. We evaluate a miner in terms of the policy it produces and later in terms of the number of bugs found by the that policy. When comparing miners we abbreviate our miner (defined in Section 3) by WN.

The first experiment compares miner performance on policies governing `hibernate2`'s `SessionFactory`, `Session` and `Transaction` classes, as described in Section 2. This example was chosen because one policy for it is clearly described in the documentation, and also because that policy is complex enough that none of the miners can expect to learn it perfectly (e.g., our technique is unable to find all of the pieces of the full specification because of its assumptions about run-time errors). ECC and our technique both find policies about these classes (and others) automatically. For the purposes of comparison, however, we restrict all miners to policies about these three classes. For Strauss, WML and JIST we also provide all of the appropriate parameters (e.g., class names, predicates).

For the purposes of the comparison we present the same raw trace data to each algorithm that looks at client code. In addition, some amount of human help was given to every miner. For ECC and WN, two of the top seven candidate policies were manually selected. For Strauss and WML-dynamic, a slice or core of

the learned policy was selected. For JIST and WML-static, all relevant predicates and fields were given.

5.1 `Hibernate2` `Session` specifications

Strauss, WML-dynamic, ECC, and our technique all learned policies similar to the documentation-based policy shown in Figure 1.

The Strauss policy (Figure 3) captures the beginning and the end of the Figure 1 closely but is less precise than Figure 1 in the middle. Strauss's use of frequency information means that common sequences of events like `find` and `delete` are included as part of the policy. Paths through states 2–6 are all particular instantiations of the "do some work" state 3 in Figure 1. Compared to Figure 1, a sequence of two `flush` events after an `openSession` is incorrectly rejected by the Strauss policy while a sequence that has `beginTransaction` but no `rollback` or `commit` is incorrectly accepted.

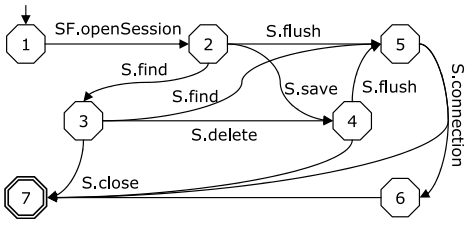

Fig. 3. A slice (the "hot core") of the `Session` policy learned by **Strauss**: the full learned specification has 10 states and 45 transitions.

The WML-dynamic policy permissively accepts all of the input traces. A slice is shown in Figure 4, the full policy has 27 states and 117 transitions. The slice captures the highlights of Figure 1 (e.g., in states 1–2–3–5–6) but fails to reject observed illegal behavior (e.g., forgetting `close`) and rejects unobserved legal behavior (e.g., `reconnect` followed by `close`). WML-dynamic makes a strong frequency assumption: a transition is valid if and only if it is observed. By contrast, our algorithm's `ex` and `oe` filters rule out some observed illegal behavior.

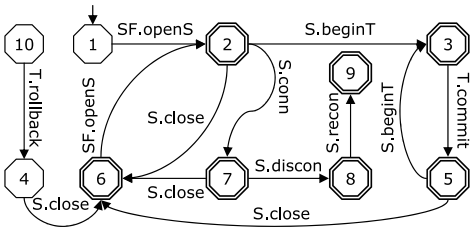

Fig. 4. A slice of the `Session` policy learned by **WML-dynamic**.

Figure 5 shows the top seven policies for these classes learned by ECC and our approach. ECC learned 350 such candidate policies. The z-statistic favors frequent pairs: the pair ⟨`beginTransaction`, `save`⟩ occurs on more than 1,500 traces, and is thus a common practice, but is not strictly required. Our approach learned 15 candidate policies, of which 2 are real. Two of the three main aspects of the documented specification, ⟨`openSession`, `close`⟩ and ⟨`beginTransaction`, `commit`,⟩ appear as #3 and #6 on the list. Since we explicitly look only for pairs ⟨a, b⟩ that occur in almost all normal traces we will not find the `rollback` policy (no normal traces include `rollback` events).

ECC Policies					**WN** Policies				
#	z-rank	Event a	Event b	Real	Rank	Event a	Event b	Real	ECC
1	9.896	S.beginTrans	S.save	Yes	1.000	S.iterate	S.close	no	286
2	1.686	S.reconnect	S.load	no	1.000	S.getIdentifier	S.close	no	28
3	1.634	S.getLockMode	S.close	no	0.971	SF.openSession	S.close	Yes	256
4	0.609	SF.openConn	SF.closeConn	Yes	0.971	S.createQuery	S.close	no	269
5	0.430	S.disconnect	S.reconnect	no	0.969	S.find	S.close	no	290
6	0.309	S.getLockMode	S.load	no	0.966	S.beginTrans	T.commit	Yes	175
7	-0.430	S.disconnect	S.load	no	0.966	S.beginTrans	S.close	no	254

Fig. 5. The top seven `Session` policies learned by **ECC** and **WN**. Each policy requires an instance of "Event a" to be followed an instance of the corresponding "Event b". The "Real" column indicates whether the specification is decidedly a false positive (no) or possibly valid (Yes). For a WN policy, the "ECC" column shows the ranked number (out of 350, low represents a likely specification) given to that policy by the ECC algorithm.

5.2 `Hibernate2` `Session` typestate specifications

The `hibernate2` documentation mentions one notion of `Session` typestate. The code does contain defensive programming checks using this typestate that raise exceptions, just as WML-static and JIST assume. Unfortunately, neither WML-static nor JIST are able to learn this typestate because it is checked by verifying that an instance object is in a dynamic data structure kept at run-time. In addition, no check raises an exception if `close`, `commit` or `rollback` are forgotten, and in general inspecting library code will miss policies about such methods, so WML-static and JIST cannot learn the full specification in Figure 1.

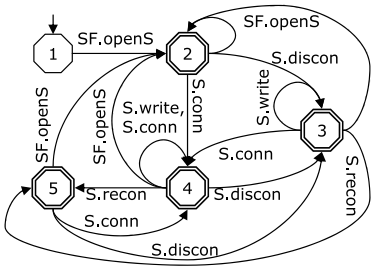

Fig. 6. `Session` policy learned by **WML-static**.

WML-static (Figure 6) discovers five illegal sequences of `Session` methods. It finds a useful undocumented `Session` typestate: two variables track the state of a `Session` as it connects to a database. The `S.write` method checks these underlying typestate variables but does not set them. For WML-static and JIST, all unlisted method invocations (e.g., `S.close`) are orthogonal to the learned policy.

JIST (Figure 7) produces a more precise policy (e.g., it discovers that `connection` cannot be followed by `writeObject`) because it does not require methods to have a uniform impact on the object's typestate. Each state in the JIST policy represents a distinct valuation of two variables. The `writeObject` method may only be called when both are false. The `reconnect` method always sets the second to true, so both techniques discover that it cannot be followed by `writeObject`. The `connection` method, however, has a different effect on the

state variables depending on their current values, so WML-static cannot reason precisely about it.

In our experiments the important difference between JIST and WML-static was not JIST's greater dataflow precision but JIST's more accurate characterization of interesting traces. All of the data manipulation was either too complicated for both methods to model (e.g., in heap data structures) or simple enough to meet WML-static's assumptions (e.g., comparing a fields and constant values). These observations support our algorithmic design choice to use simple a dataflow requirement but to pay careful attention to characterizing exceptional traces.

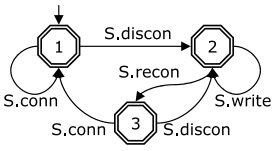

Fig. 7. `Session` policy learned by **JIST**.

6 Bug Finding via Specification Mining

We present bug-finding effectiveness results comparing (1) the performance of all algorithms on the `Session` policy and (2) our algorithm against ECC and generic "library" specifications for one million lines of code.

6.1 Comparison with Other Specification Miners

Given a candidate policy, we use an ESP-like tool [6] to find potential bugs by checking the policy against the source code [4, 6, 8, 10, 16]. Each potential bug is classified as a false positive or a real error by manual inspection. For example, if an application fails to close a file but immediately shuts down as a result of the error, the "leaked" file is classed as a false positive. However, a leaked database lock between the JVM (held on behalf of the program) and an external database is a bug if no finalizers close the connection when the program (but not the JVM) shuts down.

Mining Technique	False Positives	Real Errors
Strauss-Full	27	0
Strauss-Cored	20	46
ECC #1	30	20
ECC #4	1	0
WN #3	4	46
WN #6	3	20
WML-static	9	0
JIST	1	0

Fig. 8. Comparison of miner bug-finding power for `hibernate2 Session` policies. "False Positives" are methods that violate the mined policy but are actually correct. "Real Errors" are buggy methods that violate the mined policy.

In Figure 8 we present the results of using the mined specifications to find bugs in the `hibernate2` program. Each "false positive" or "real error" represents a method where a trace fails to adhere to the given policy. The WML-dynamic approach is not shown because its specification accepts all of the traces by construction (thus it finds no bugs but yields no false positives).

Strauss-Full, the entire 10-state policy learned by Strauss, yields too many false positives to be effective for bug-finding. Twenty-five of the false positives are from traces along which `S.close` occurs after a sequence of "work" that the specification fails to accept. However, since the specification also has many accepting states (in particular, the state after `SF.openSession` accepts), errors involving forgetting `S.close` are not reported.

Strauss-Cored, the sliced policy shown in Figure 3, gives a reduced number of false positives compared to Strauss-Full, but still suffers from the same problems. However, Strauss-Cored is able to find 46 methods in which `openSession` is called but `close` is not (and 4 false positives involving `openSession`).

ECC, using specification #1 (the policy with the highest z-rank, see Figure 5), finds 20 methods that deal with `beginTransaction` improperly, 3 false positives involving `beginTransaction` and 27 false positives involving `save`. ECC specification #4 turns out not to be useful for bug finding. Its z-rank is high (28 of 30 traces that mention a also mention b), but it only occurs at one point in the source code. Either the $z\text{-rank}|_N$ or our ranking would rank it much lower ($N_a = 1, N_{ab} = 1$).

Our method using specification #3 finds all 46 of the `Session` leaks found by Strauss-Cored (and the same four false positives). In fact, the Strauss-Cored report is a superset of the WN #3 report. Using specification #6 we are able to find the 20 methods with `commit` and `rollback` mistakes that are also found by ECC. Along 20 of the 23 error paths we report in which `beginTransaction` occurs but `commit` does not, `rollback` does not either. The ECC #1 report is a superset of the WN #6 report (but with additional false positives).

Neither the WML-static nor the JIST specification lead to the discovery of any bugs in this example. No traces contain `S.discon` followed by `S.discon`, for example (or indeed any other erroneous violations of this typestate specification). The JIST specification yields fewer false positives because it more accurately represents the underlying `Session` typestate.

We conclude from these experiments that (1) the various techniques produce different kinds of specifications, in accordance with their assumptions about how programmers make mistakes and (2) not all of the assumptions underlying these miners were born out by this example (such as the assumption that typestate would be explicitly and simply represented or assumptions about event frequency). WML-static and JIST were both able to find an undocumented typestate specification. Their low false positive count shows that they were able to form specifications that were permissive enough to accept most client behaviors. Strauss, ECC and our technique were all good at yielding specifications that found bugs. Our technique found all bugs reported by other techniques and did so with the fewest false positives.

6.2 Larger Programs and Candidate Specification Ranking

Figure 9 compares our technique and the ECC technique on various benchmarks. The benchmarks were chosen for ease of comparison with previous work, and may favor the "a must be followed by b" specifications that both WN and ECC are

Program	Lines of Code	WN (our miner) Real Specs	WN (our miner) Bugs via Specs	ECC Real Specs	ECC Total Specs	ECC Bugs via Specs	Library Policy Bugs
`infinity 1.28`	28k	1 / 10	4	0 / 227	6468	0	14
`hibernate2 2.0b4`	57k	9 / 51	93	3 / 424	9591	21	13
`axion 1.0m2`	65k	8 / 25	45	0 / 96	4159	0	15
`hsqldb 1.7.1`	71k	7 / 62	35	0 / 224	5032	0	18
`cayenne 1.0b4`	86k	5 / 35	18	3 / 311	8432	8	17
`sablecc 2.17.4`	99k	0 / 4	0	0 / 80	2506	0	3
`jboss 3.0.6`	107k	11 / 114	94	2 / 444	12852	4	40
`mckoi-sql 1.0.2`	118k	19 / 156	69	2 / 346	10860	5	37
`ptolemy2 3.0.2`	362k	9 / 192	72	3 / 656	23522	12	27
total	993k	69 / 649	430	13 / 2808	83422	50	172

Fig. 9. Bugs found with specifications mined by ECC and our technique. The "Real Specs" column counts valid specifications (determined by manual inspection) against candidate specifications. For WN, all candidate policies were inspected. For ECC, only candidates with non-negative z-rank were inspected. The "Total Specs" column counts all policies reported by ECC. The "Bugs via Specs" column counts methods that violate the "Real Specs". Finally, the last column counts methods violating a generic "library"-based policy that was applied equally to all programs.

designed to mine. We also compare the bugs found via specification mining to the bugs found via the generic "library" specifications we used in our previous work [16]. The library policies were two- to four-state FSMs describing network connections, database locks and file handles. We are unable to directly compare the other techniques because of the cost involved in manually specifying classes, predicates, and other parameters in advance.

ECC is able to find 4 specifications missed by our algorithm. In one of these examples, the b event never occurs in any error handling code (and thus does not meet our `ex` requirement). Removing the `ex` requirement causes our algorithm to produce 1,114 candidate specifications for `hibernate2` alone. Given the small number of real specifications that are filtered by the requirement and the large number of false positives that it avoids, we believe that basing our algorithm on exceptional control flow paths was a good decision.

Of the 69 real specifications we found, 24 involved methods from separate classes, arguing against class-based module requirements. Only one valid specification involved methods from different libraries. On the other hand, for example, 30 of the first 100 false positive specifications reported by ECC for `axion` could have been avoided with our `sp` package-level module requirement. We believe these results argue strongly in favor of package-level requirements.

A common false positive for ECC paired the family of methods `ListIterator.hasNext` and `ListIterator.next`. The vast majority of paths that contain the former also contain the latter, and iterators occur frequently, causing the z-rank (whether restricted to normal traces or not) for such pairs to be high (`iterator` specifications occur as one of the top five candidates for ECC on six of our nine programs).

A common false positive for our technique paired `read` or `write` (instead of `open`) with `close`. As the ⟨`flush`, `close`⟩ data in Figure 2 demonstrate, "intermediate" work functions like `read` are almost invariably followed by `close` if they are present, but the more desirable `open`-based specification usually ranks higher.

Almost every valid specification our technique found was listed somewhere in ECC's output of candidate specifications. For example, our 59th candidate `jboss` policy finds four real errors and is #9522 on the ECC list (z-rank$= -54$, z-rank$|_N = -29$).

Figure 10 shows the number of bugs found as a function of the ranking used to sort candidate specifications produced by our algorithm. Compared to the z-rank, our ranking only required 42% of the specifications to be inspected (instead of 72%) in order to find two-thirds of the bugs. However, we conclude that since various rankings work only moderately better than a random shuffle, it is very important to produce a small number of extraneous candidates.

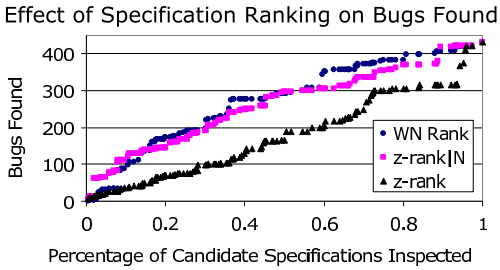

Fig. 10. Bugs found as a function of the rank order in which candidate specifications are inspected. "WN Rank" is $N_{ab}/(N_a+N_{ab})$, the ranking used by our algorithm, "z-rank$|_N$" is the z-statistic restricted to normal traces and "z-rank" is the z-statistic.

Our results for ECC are consistent with, but slightly better than, previously published figures in which 23 errors were found via specification mining on the Linux 2.4.1 kernel (about 840k LOC) [8]. ECC was designed to target C operating systems code. It actually performs better (in errors found per line of code) in this domain than in their reported experiments, although there is no reason to believe that the bug density should be the same.

One additional consideration is the utility of the found bugs. Evaluating the importance of a bug is beyond the scope of this work. Our mining technique favors resource leaks and forgotten obligations. One of the authors of `ptolemy2` was willing to rank resource leaks we found on his own scale. For that program, 11% of the bugs we reported were in tutorials or third-party code, 44% of them rated a 3 out of 5 for taking place in "little used, experimental code", 19% of them rated a 4 out of 5 and were "definitely a bug in code that is used more often", and 26% of them rated a 5 out of 5 and were "definitely a bug in code that is used often." We cannot claim that this breakdown generalizes, but it does provide one concrete example.

Using our mining algorithm to find bugs was decidely better than using generic library policies. We found 430 bugs using mined policies compared to 172 using generic ones. We found 380 more bugs and 56 more policies than ECC using 2000 fewer candidate specifications. Our experiments highlighted the

practical importance of our algorithmic assumptions, in particular our use of exceptional control flow.

7 Conclusions

As automatic program verification tools become more prevalent, specifications become the limiting factor in verification efforts, and specification mining for the purposes of finding bugs becomes more important. Given a program, a specification miner emits candidate policies that describe real or common program behavior. We propose a novel miner that uses information about exceptional paths. We compare the bug-finding power of various miners. In 1 million lines of Java code, we found 430 bugs using mined specifications compared to 172 using generic "library"-based ones, and we found more bugs than comparable mining algorithms. Our experiments highlighted the relative unimportance of candidate ranking and the practical importance of our algorithmic assumptions, in particular our use of exceptional control flow for specification mining.

8 Acknowledgments

We are indebted to Ras Bodik, Glenn Ammons and Dave Mandelin for insightful discussions and for setting up Strauss for our experiments. We thank Dawson Engler for enlightening discussions about his technique, z-ranking, and expected results. We thank John Whaley for providing us with the `joeq` source code and pointers for running WML. Finally, we thank Rajeev Alur for providing a number of examples of JIST in action.

References

1. R. Alur, P. Cerny, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. In *ACM Symposium on Principles of Programming Languages*, 2005.
2. G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *ACM Symposium on Principles of Programming Languages*, pages 4–16, 2002.
3. G. Ammons, D. Mandein, R. Bodik, and J. Larus. Debugging temporal specifications with concept analysis. In *Programming Language Design and Implementation*, San Diego, California, June 2003.
4. T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001, Workshop on Model Checking of Software*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122, May 2001.
5. H. Chen, D. Dean, and D. Wagner. Model checking one million lines of C code. In *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2004.
6. M. Das, S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. *SIGPLAN Notices*, 37(5):57–68, 2002.

7. R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *ACM Conference on Programming Language Design and Implementation*, pages 59–69, 2001.
8. D. R. Engler, D. Y. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Symposium on Operating Systems Principles*, pages 57–72, 2001.
9. D. Freedman, R. Pisani, and R. Purves. *Statistics*. Third edition. W. W. Norton, 1998.
10. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *ACM Symposium on Principles of Programming Languages*, pages 58–70, 2002.
11. Hibernate. Object/relational mapping and transparent object persistence for Java and SQL databases. In *http://www.hibernate.org/*, July 2004.
12. K. R. M. Leino and G. Nelson. An extended static checker for Modula-3. In K. Koskimies, editor, *Compiler Construction*, volume 1383 of *Lecture Notes in Computer Science*, pages 302–305, Apr. 1998.
13. B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Programming Language Design and Implementation*, San Diego, California, June 9–11 2003.
14. G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *ACM Symposium on Principles of Programming Languages*, pages 128–139. ACM, Jan. 2002.
15. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.
16. W. Weimer and G. Necula. Finding and preventing run-time error handling mistakes. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Vancouver, British Columbia, Canada, Oct. 2004.
17. J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *International Symposium of Software Testing and Analysis*, 2002.