Automatic Mining of Specifications from Invocation Traces and Method Invariants

Ivo Krka*
Google Inc.
Zurich, Switzerland
krka@google.com

Yuriy Brun
University of Massachusetts
Amherst, MA, USA
brun@cs.umass.edu

Nenad Medvidovic
University of Southern California
Los Angeles, CA, USA
neno@usc.edu

ABSTRACT

Software library documentation often describes individual methods' APIs, but not the intended protocols and method interactions. This can lead to library misuse, and restrict runtime detection of protocol violations and automated verification of software that uses the library. Specification mining, if accurate, can help mitigate these issues, which has led to significant research into new model-inference techniques that produce FSM-based models from program invariants and execution traces. However, there is currently a lack of empirical studies that, in a principled way, measure the impact of the inference strategies on model quality. To this end, we identify four such strategies and systematically study the quality of the models they produce for nine off-the-shelf libraries. We find that (1) using invariants to infer an initial model significantly improves model quality, increasing precision by 4% and recall by 41%, on average; (2) effective invariant filtering is crucial for quality and scalability of strategies that use invariants; and (3) using traces in combination with invariants greatly improves robustness to input noise. We present our empirical evaluation, implement new and extend existing model-inference techniques, and make public our implementations, ground-truth models, and experimental data. Our work can lead to higher-quality model inference, and directly improve the techniques and tools that rely on model inference.

Categories and Subject Descriptors:
D.2.5 [Testing and Debugging]: Debugging aids, Tracing
General Terms: Algorithms, Design, Modeling
Keywords: Model inference, execution traces, log analysis

1. INTRODUCTION

Developers frequently use existing software libraries. Unfortunately, many libraries are poorly documented, hindering reuse and forcing developers to re-engineer existing solutions [23, 51]. Even heavily-used, well-documented libraries contain documentation inaccuracies as updates to the documentation lag or fail to follow code updates [14, 54]. Further, natural-language documentation can be ambiguous and misunderstood by developers [13], causing library misuses that lead to subtle, latent, and costly faults. A promising way to ease library use is to automatically mine accurate specifications from existing uses of those libraries [59]. Numerous specification mining techniques have been proposed [2, 3, 4, 11, 17, 20, 21, 24, 39, 41, 43, 52, 58] and used for test case generation [16, 41], debugging [4], validation [18], and fault localization [48].

The quality of the inferred specifications is critical to their successful use. In fact, most research on specification inference focuses on improving the quality of the inferred specifications. Yet, the relationship between the principles employed by inference techniques, and the quality of the models they produce is not well understood, and existing studies have not pursued to improve this understanding in the context of real-world software. The goal of this paper is to systematically evaluate the principles behind existing inference techniques, particularly with respect to the input data used by the techniques, to understand the effects of these principles on the inferred models' quality and on the scalability of the inference. The main contributions of this work are (1) an enhanced understanding of how to best design and improve inference techniques, and (2) a novel technique that implements these improvements.

Existing mining techniques either (1) infer finite state machine (FSM) models that support the observed invocation sequences (also referred to as *execution traces*) [2, 4, 10, 21, 39, 43, 49, 56], or (2) identify high-level properties — declarative class and method invariants — by observing how a library's state (its internal variables) changes at runtime [11, 20, 57]. Hence, the utility of the dynamic inference techniques in general, and FSM-inference techniques in particular, critically depends on how rich the inferred models are and how close they are to the true model of the system's behavior. The primary focus of FSM-inference research has been to improve the inferred models' precision (e.g., [10, 39, 41, 49, 56]). Recent research suggests that the inferred models' quality can be improved by extending the FSM inference that relies solely on execution traces, by also using internal state information available during execution [16, 41]. In the limit, the Contractor algorithm demonstrates that FSM inference can rely exclusively on internal state information [18].

While existing evaluations of model inference research have made substantial contributions, they also have had significant limitations our work addresses. Most importantly, to properly compare inference techniques, it is necessary to understand how well they perform on real software, and how their underlying principles (e.g., reliance on inferred state invariants vs. reliance on execution traces) affect model quality. Many existing studies of model quality rely on simulated execution traces, as opposed to traces from actual software systems [10, 36, 39]. As simulations are typically more controlled than real software, this runs the risk of inaccurately esti-

*This work was completed while Ivo Krka was a PhD student at the University of Southern California.

FSE '14, November 16–22, 2014, Hong Kong, China
ACM 978- 1-4503-3056-5/14/11
http://dx.doi.org/10.1145/2635868.2635890

mating how the techniques perform in the wild. Meanwhile, those studies performed on real software typically estimate the quality of the inferred models via proxies. For example, the quality is sometimes measured via test coverage [41], the number of automatically detected faults [16, 21], or case-study analysis of manually discovered faults [3, 4, 18], instead of the more generalizable information retrieval metrics of precision and recall [36].

Further, there are no studies to date that thoroughly measure the effects of the types of input information used by inference techniques on the quality of the resulting models. In particular, there are four possible strategies to dynamic FSM inference:

1. **Traces-only**: infer models from execution traces only.
2. **Invariants-only**: infer models from invariants.
3. **Invariant-enhanced-traces**: infer models from execution traces and then enhance them with invariants.
4. **Trace-enhanced-invariants**: infer models from invariants and then enhance them with execution traces.

In this context, the research questions "How do these strategies impact the quality of the inferred models?" and "Under what circumstances are they effective?" remain unanswered. (Notably, the empirical study conducted by Lo et al. [40] provides an initial comparison of the first and the third strategies above.)

To answer these questions, this paper provides an empirical study that compares the quality of models inferred by four techniques that, respectively, implement the above four strategies. The k-tail [5] inference algorithm, representing the traces-only strategy, is employed and enhanced by many model-inference techniques (e.g., [2, 3, 39, 41, 43, 49]). We select k-tail because k-tail models turn out to already be highly precise in our evaluations, and further precision improvements would not affect our findings. CONTRACTOR++, representing the invariants-only strategy, is our own extension of the Contractor technique [18] that creates models based solely on the inferred state invariants. CONTRACTOR++ addresses the limitations of Contractor when applied to dynamically inferred, as opposed to manually specified, state invariants. SEKT, representing the invariants-enhanced-traces strategy, is our own extension of k-tail that uses state invariants to restrict the way k-tail merges execution traces. SEKT expands on Lorenzoli et al.'s *gk-tail* algorithm [41] by relaxing restrictions on how and which invariants are inferred for small subsets of the execution traces. Finally, TEMI, representing the trace-enhanced-invariants strategy, is a novel technique we have developed as part of this research that first infers a model based on the state invariants [20], and then refines that model with the information from the execution traces.

A systematic, careful evaluation of the four strategies has two critical requirements. First, the findings must be generalizable. Second, they must be applicable to real-world systems. Our evaluation relies on nine widely-used, open-source libraries selected from a range of domains. Further, our evaluation derives the execution traces by running eight real-world, publicly-available applications that use the selected libraries. By doing so, our evaluation reflects more closely than most existing studies the scenario of an engineer encountering a poorly documented library and trying to infer a usage model from other open-source software uses of that library. We compare the quality — represented by precision and recall — of the models produced by the four strategies. We also assess two important and often overlooked aspects of model inference: the scalability of the strategies and their robustness to noise in the inputs.

We make four evidence-supported conclusions:

1. Invariants-only and trace-enhanced-invariants strategies produce significantly higher recall than the traces-only and invariant-enhanced-traces strategies, while maintaining (or, in rare cases, minimally reducing) the already high precision.
2. In the general case, invariant-enhanced-traces and trace-enhanced-invariants strategies that combine the two types of execution information slightly improve the precision of the inferred models, while maintaining the recall.
3. Invariant filtering that keeps only a limited set of relevant invariant types is crucial to enhancing the scalability of techniques that implement invariants-only and trace-enhanced-invariants strategies.
4. While the quality of models inferred by invariants-only and trace-enhanced-invariants strategies is similar in most cases, combining an FSM inferred from invariants with execution traces circumvents the risks associated with noisy invariants. Such noise significantly reduces the quality of invariants-only strategy models, making it also perform worse than the traces-only and invariant-enhanced-traces strategies.

The remainder of the paper is organized as follows. Section 2 overviews the background. Section 3 details the techniques representing the four strategies. Section 4 evaluates those techniques, while Section 5 discusses the impact of the results. Section 6 places our work in the context of related research. Finally, Section 7 summarizes the paper's contributions.

2. BACKGROUND

This section provides the background necessary for the discussions in this paper. Section 2.1 introduces `StackAr`, our running example data structure. Section 2.2 defines the MTS formalism used by three of the model-inference techniques in Section 3 and discusses how execution traces map to an MTS. Finally, Section 2.3 defines program state and discusses state invariants.

2.1 Example Library: StackAr

`StackAr` is a Java implementation of a stack, distributed with Daikon [15, 20]. Initialized with an integer capacity, `StackAr` has six public methods: `push(x)`, `top()`, `topAndPop()`, `makeEmpty()`, `isEmpty()`, and `isFull()`. Internally, `StackAr` represents its stack as an array (`theArray`) and has a pointer to the top of the stack (`topOfStack`). A `push()` on a full stack generates an exception. A `top()` or `topAndPop()` on an empty stack returns `null`.

2.2 Modal Transition System (MTS)

An MTS [33] is an FSM-based model with labeled transitions between states. A state represents a specific point in the execution of a system or module; a transition represents the system's change from one state to another, caused by some invocation. In an MTS, there are two explicit kinds of transitions: *required*, which are transitions that are certain to occur and are common to all FSMs, and *maybe*, which are uncertain transitions unique to MTSs. We use the notation $s \xrightarrow{l}_r s'$ and $s \xrightarrow{l}_m s'$, respectively, to denote required and maybe l-labeled transitions between states s and s'.

The most common input to a model-inference algorithm is a set of observed execution traces. An execution trace — a runtime recording of public method invocations and internal data values between those invocations — can be represented by an MTS with states corresponding to variable values and transition labels composed of the method name, input values, and return value.

2.3 Program State and State Invariants

A program's concrete state is represented by the values of the program's variables at a given snapshot in the program's execution. However, for non-trivial programs, there exist intractably many concrete states. For example, for `StackAr`, there are an infinite

```
DataStructures.StackAr:::CLASS
 this.topOfStack >= -1
 this.topOfStack <= size(this.theArray[])-1

DataStructures.StackAr.push(java.lang.Object):::ENTER
 this.topOfStack < size(this.theArray[..])-1
 this.topOfStack >= -1

DataStructures.StackAr.push(java.lang.Object):::EXIT103
 orig(this.topOfStack) < size(this.theArray[..])-1
 this.topOfStack >= 0
 this.topOfStack - orig(this.topOfStack) - 1 == 0
 size(this.theArray[..])  == orig(size(this.theArray[..]))
```

Figure 1: A subset of Daikon's program invariants on `StackAr`.

number of concrete states represented by the contents of `theArray`. Therefore, it is common to consider and reason about abstract program states [17, 62] defined with first-order predicates over program variable values: Different program states correspond to different combinations of predicate evaluations. For `StackAr`, one reasonable predicate that can be used to define abstract state is "is the stack not full?" (`topOfStack` $<$ `size(theArray)` -1).

Program-state invariants can be used to automatically extract relevant predicates. Invariants hold true at certain program execution points. For example, an object-level invariant, such as `size` $\geq$ `0` holds at all program points. While developers can manually specify program invariants in the code or in other documentation [18, 46], static and dynamic analyses can automatically infer invariants.

A commonly used tool for automatic invariant inference is Daikon [20], which observes data values of program executions and infers invariants that hold over all observed executions. The inferred invariants consist of method pre- and postconditions and object invariants. Figure 1 shows eight invariants Daikon infers for `StackAr`: two object-level invariants, and two preconditions and four postconditions for `push()`. As preconditions and object invariants help determine which methods can execute in a particular state, the predicates that appear in Daikon-inferred preconditions and object invariants are good candidates for defining abstract program state. For `StackAr`, Daikon reports four predicates:
$P_1 = (\texttt{topOfStack} \geq -1)$,
$P_2 = (\texttt{topOfStack} \geq 0)$,
$P_3 = (\texttt{topOfStack} < \texttt{size(theArray)} - 1)$, and
$P_4 = (\texttt{topOfStack} \leq \texttt{size(theArray)} - 1)$.
Four boolean predicates can define $2^4 = 16$ abstract program states. However, many of these states can be automatically eliminated because of predicate interdependencies. For example, `StackAr`'s P_2 cannot be true when P_1 is false.

KTAIL(trace-set *traces*, int k)
1 MTS T = PTAI(*traces*)
2 **for each** pair $T.s_1, T.s_2$ in $T.S$
3 **if** TAIL($T.s_1, k$) = TAIL($T.s_1, k$)
4 MERGE($T.s_1$, $T.s_1$)
5 **return** T

Figure 2: The k-tail algorithm.

3. INFERENCE ALGORITHMS

This section details the four algorithms that respectively implement each the four model inference strategies (recall Section 1). k-tail (Section 3.1) implements the *traces-only* strategy by reasoning exclusively about the invocation sequences observed in the execution traces. Contractor [18] (Section 3.2) builds a model by reasoning about the potentially allowed invocation sequences based on manually specified state invariants. To apply Contractor on dynamically inferred invariants, we have developed CONTRACTOR++ via a set of non-trivial extensions of Contractor. SEKT (Section 3.3) is our extension of k-tail inference that implements the invariant-enhanced-traces strategy by considering the program states extracted from invariants as a k-tail merging criterion. Finally, TEMI (Section 3.4) implements the trace-enhanced-invariants strategy by first building an MTS that adheres to the invariants and then refining it according to the execution traces.

3.1 Traces-Only: Traditional k-tail

The existing k-tail algorithm [5] concisely captures a library's API protocol as an MTS with required-only transitions by merging the states from the execution traces. The algorithm merges every pair of states with identical sequences of the next k invocations (hence "k-tail"). Figure 2 details the algorithm. In the first step (line 1), k-tail builds a prefix-tree-acceptor (PTA). The PTA is an MTS obtained by merging the initial states of each trace into a single initial state, and from that state, merging all those states that are reached by the same invocation sequence (e.g., states s_1 through s_4 of Trace 1, are merged with states s_1 through s_4 of Trace 4, respectively). Subsequently, lines 2–4 analyzes each pair of states in the PTA for tail equivalence; line 4 merges the equivalent states.

The k-tail algorithm relies on selecting an appropriate k. This choice typically involves a tradeoff between precision (smaller k implies more spurious merges due to the limited scope) and completeness/recall (larger k implies fewer merges and less generalization) of the generated model. Because of that, the existing k-tail-based techniques (e.g., [39, 41, 49]) have aimed to improve the algorithm's precision with smaller k. Despite some improvements, the existing k-tail-based algorithms still suffer from limitations described next.

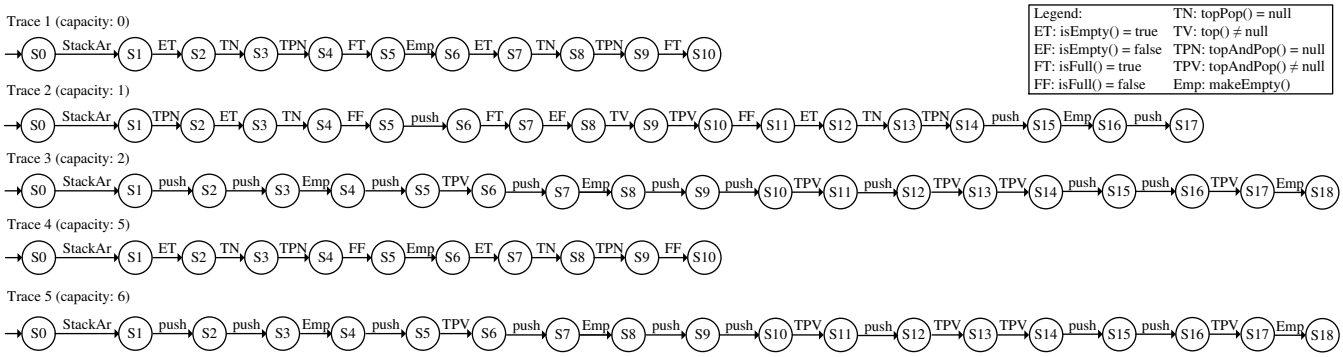

Figure 3: Five example `StackAr` invocation traces.

```
CONTRACTOR(inv-set invariants)
1  MTS T, T.S = ∅
2  for each tuple enabled ∈ invariants.methods
3    clause statePred = ∅
4    for each method ∈ enabled
5      statePred = statePred ∧ method.preCond
6    for each method ∉ enabled
7      statePred = statePred ∧ ¬method.preCond
8    if SMT-ISCONSISTENT(statePred)
9      add T.s_enabled to T.S
10 for each pair T.s1, T.s2 in T.S and method in inv.methods
11   if SMT-ISCONSISTENT(T.s1 ∧ method.preCond)
12   and SMT-ISCONSISTENT(T.s2 ∧ method.postCond)
13     add T.s1 --method-->_r T.s2 to T.transitions
14 return T
```

Figure 4: The Contractor algorithm.

To illustrate k-tail and its shortcomings, we use five `StackAr` invocation traces, corresponding to creating and using stacks of different capacities (Figure 3). Let us consider how 2-tail that considers the method return values as part of its merging criterion works on these traces. The algorithm correctly merges states s_1 and s_6 in Trace 1 because the two following invocations from each state are `isEmpty()=true` and `top()=null`. The algorithm also merges states s_1 in Trace 1 and s_{11} in Trace 2. However, this merge is imprecise as it allows a non-zero-capacity stack to change capacity to zero after an invocation of `isFull()` from s_{10} to s_{11} in Trace 2. The k-tail models are also incomplete for those traces in which pairs of methods happen to be, but are not actually required to be, invoked in a specific order (e.g., `top()` is always invoked after `isEmpty()` in Figure 3). The existing k-tail-based techniques (e.g., [39, 41, 49]) all suffer from these limitations.

3.2 Invariants-Only: CONTRACTOR++

As mentioned earlier, CONTRACTOR++ comprises two parts: (1) the core algorithm Contractor [18], a recent algorithm that creates MTS models exclusively based on program invariants, and (2) our enhancements to Contractor take enable it to work on inferred, as opposed to manually specified, program invariants.

Figure 4 lists the Contractor algorithm. Contractor uniquely characterizes the model's states by the combination of methods enabled in each state (lines 2–9 build the state set $T.S$). This abstraction is thus referred to as *enabledness*. A state (i.e., a combination of enabled methods) is legal if the preconditions of the enabled methods are consistent with one another. The algorithm checks for consistency of the preconditions in lines 2–9 with the help of an off-the-shelf Satisfiability Modulo Theories (SMT) solver, such as Yices [61]. Lines 10–14 create a transition on a method between two states if that method's precondition is satisfied in the source state and the postcondition is satisfied in the target state.

To use Contractor in our evaluation, we had to enhance its inputs in two significant ways, without which both the accuracy and scalability of the algorithm would be notably lower. We refer to these enhancements as CONTRACTOR++, and use CONTRACTOR++ as the implementation of the invariants-only inference strategy.

We refer to the first enhancement as *Method Distinction*: Since Contractor does not consider predicates of the methods' output values, in CONTRACTOR++ we represent each *(method, return-value)* combination as a distinct method with its own invariants. Otherwise, for example, the model inferred for `StackAr` would have only two states: the first one with all methods enabled, and the second one with `push()` prohibited and all other methods allowed. Such a model is imprecise as it allows, for example, pushing on a stack of 0-capacity, or popping non-null values from an empty stack.

```
ADVANCEDMERGE(st T1.S, st T2.P, int k, set pred)
1  if Tail(T1.S, k) ≠ Tail(T2.P, k)
2    return false
3  if EVALGLOBAL(T1.S, pred) ≠ EVALGLOBAL(T2.P, pred)
4    return false
5  return true
```

Figure 5: The SEKT algorithm uses ADVANCEDMERGE to determine if two states should be merged.

We refer to the second enhancement as *Invariant Filtering*: Instead of using manually-specified invariants, CONTRACTOR++ uses Daikon-inferred invariants, filtered to avoid less meaningful invariants and to make the algorithm scale, since dynamically inferred invariants typically have higher complexity than the manually specified ones [46]. We consider the relational invariants on boolean and integer variables (e.g., `IntEqual`, `IntGreaterThan`), and the `IsNull` invariant on objects. Further, we consider internal variables up to a depth of one (i.e., we consider an object's fields, but not those fields' fields). For collections, we consider their sizes but not their elements. Without these enhancements, the original Contractor algorithm's models are of significantly lower quality than reported below for CONTRACTOR++. Note that the same set of filters are employed in (and were, in fact, originally developed for) TEMI.

3.3 Invariant-Enhanced-Traces: SEKT

While k-tail has been used extensively in prior work (recall Section 3.1), our State-Enhanced k-tail, SEKT, is the first algorithm that extends k-tail using program state information inferred from the full set of observed executions. The most closely related algorithm, proposed by Lorenzoli et al. [41], infers invariants only for the limited set of states that are merged during PTA construction (recall Figure 2). Due to this limited scope for learning relevant invariants, Lorenzoli et al.'s algorithm may err both in allowing and rejecting merges, which may decrease both the precision and the recall of the resulting model [40]. For example, this approach merges the states in `StackAr`'s Trace 1 and Trace 4 from Figure 3 because they follow the same invocation sequence, despite the different method parameter and return values. Consequently, the resulting merged trace erroneously implies that a stack of size 0 is the same as one of size 5 even though `isFull` returns different values.

In contrast to Lorenzoli et al.'s algorithm, SEKT modifies the k-tail algorithm by adding a new global merge requirement: The merging states must correspond to the same abstract program state. Method ADVANCEDMERGE in Figure 5 details this merging criterion and is incorporated into k-tail by replacing the original condition specified in line 3 of Figure 2. The *pred* parameter of ADVANCEDMERGE is the set of predicates that define the program states, and $T_1.S$ and $T_2.P$ are the two potential to-be-merged states. Lines 1–2 of ADVANCEDMERGE test the tail similarity; lines 3–4 check for matching program states. The state matching compares whether the two concrete variable evaluations correspond to the same abstract program state (EVALGLOBAL).

The new merging condition can prevent erroneous merges. For example, SEKT avoids the spurious merge of s_1 in Trace 1 and s_{11} in Trace 2 in the `StackAr` traces from Figure 3 (recall Section 3.1): The condition in line 3 of ADVANCEDMERGE is satisfied, thus rejecting the merge, since only predicates P_1 and P_4 (from Section 2.3) are true in s_1, but only P_1 and P_3 are true in s_{11}.

3.4 Trace-Enhanced-Invariants: TEMI

Trace-Enhanced MTS Inference, TEMI, infers an MTS that includes but differentiates (1) the behavior *asserted* legal by the in-

```
GENERATEINVARIANTMTS(set pred, inv-set invariants)
1  MTS invariantMTS, set toProcess, isProcessed = ∅
2  set predCombinations = COMBINE(pred)
3  for each combination ∈ predCombinations
4    if SMT-ISCONSISTENT(combination)
5      add combination to invariantMTS.states
6  add invariantMTS.initSt to toProcess
7  while toProcess ≠ ∅
8    currentSt = toProcess.pop
9    add currentSt to isProcessed
10   for each methodInv ∈ invariants
11     if SMT-ISCONSISTENT(methodInv.pre ∧ currentSt)
12       for each targetSt ∈ invariantMTS.states
13         if SMT-ISCONSISTENT(methodInv.post ∧ targetSt)
14           if targetSt ∉ isProcessed add targetState to toProcess
15           add currentSt --methodInv.name-->_m targetSt to invariantMTS.transitions
16 return invariantMTS
```

Figure 7: Constructing of an invariant-based MTS.

variants and (2) the behavior *observed* in the traces. TEMI consists of two phases. The first phase, conceptually similar to CONTRACTOR++, constructs an MTS with only maybe transitions, capturing all invocation sequences of an object's interface allowed by the invariants. We call this model an invariant-based MTS. The second phase promotes transitions observed in the traces from maybe to required. The remaining maybe transitions stem from generalizations performed during invariant inference. As we demonstrate later in the paper, these generalizations are error-prone when working with a partial, limited set of execution traces. TEMI is loosely inspired by our earlier work on refining requirements-level use-case scenarios [29]. We have previously outlined an early version of TEMI [31], but the algorithmic details and evaluation in this paper are new.

Phase I: Synthesis of the Invariant-Based MTS

The method GENERATEINITIALMTS (Figure 7) synthesizes an invariant-based MTS. It first constructs the prospective state space *invariantMTS.states* (lines 3–5) based on the set *pred* of predicates from method preconditions. For each possible combination of the predicate evaluations (line 2), GENERATEINVARIANTMTS uses Yices to check if that combination, in conjunction with object invariants, is satisfiable (line 4). For `StackAr`, Daikon inferred 4 predicates (recall Section 2.3); Yices rejects as unsatisfiable every predicate combination with $P_1 = \text{false}$ and $P_2 = \text{true}$.

After determining the valid states, GENERATEINVARIANTMTS creates transitions between those states (the loop in lines 7–15). Each transition added in line 15 has a source state that satisfies the appropriate method preconditions, and a destination state that satisfies the postconditions, similarly to CONTRACTOR++. The resulting invariant-based MTS contains a state for every reachable program state and a transition for every invocation sequence that is legal according to the invariants [29].

Figure 8 depicts the invariant-based MTS for `StackAr`. Although the largest theoretical state space for `StackAr` is $2^4 = 16$ states, with an additional initial state, the generated invariant-based MTS has only 5 states (including the initial state). There are several self-transitions that capture methods that do not change the program state. By contrast, the k-tail-based algorithms implicitly consider every method to be state-changing.

TEMI's construction of the invariant-based MTS is conceptually similar to CONTRACTOR++ as it uses a predicate-based abstraction of the program state. In contrast to Contractor, whose predicates are the full method preconditions, TEMI uses the individual clauses that appear in the invariants. The reason we choose this finer-grain abstraction is that the automatically inferred postcondi-

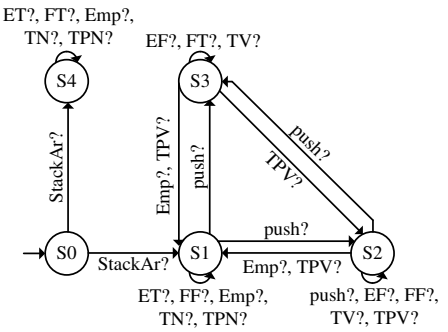

Figure 8: The invariant-based `StackAr` MTS.

```
REFINEINVARIANTMTS(MTS invMTS, set traces)
1  currentSt = invMTS.initialState
2  for each currentEv ∈ traces
3    for each nextSt ∈ invMTS.states
4      if SMT-ISCONSISTENT(nextSt ∧ currentEv.post)∧
                                  currentSt --currentEv-->_m nextSt
5        REFINESTATE(currentSt,nextSt,currentEv,invMTS)
6    currentSt = nextSt'
7  for each st1 ∈ invMTS.refinedSt
8    for each st1 --l-->_m st2 ∈ invMTS.transitions
9      if ∃st1 --l-->_r st3 remove st1 --l-->_m st2
10 for each state1,state2 ∈ invMTS.refinedSt
11 where state1.programSt = state2.programSt
12   if st1.outTrans ≈ st2.outTrans MERGE(st1, st2)
13 return invMTS

REFINESTATE(state currentSt, state nextSt, event currentEv, MTS invMTS)
1  if currentSt = nextSt require currentSt --currentEv-->_r nextSt
2  else
3    add nextSt' to invMTS.states, rename nextSt to nextSt''
4    replace currentSt --currentEv-->_γ nextSt'' with currentSt --currentEv-->_r nextSt'
5    for each nextSt'' --l-->_γ otherSt in invMTS.transitions
6      if nextSt'' ≠ otherSt add nextSt' --l-->_γ otherSt to invMTS.transitions
7      else add nextSt' --l-->_γ nextSt' to invMTS.transitions
```

Figure 9: Refining the invariant-based MTS according to the traces.

tions tend to be more complex than the manually written ones [46]. For example, consider a method postcondition that consists of a set of implication clauses that relate the program state before and after a method invocation: $[(PreState_1) \Rightarrow (PostState_1)] \wedge \ldots \wedge [(PreState_n) \Rightarrow (PostState_n)]$. The states in the TEMI's invariant-based MTS would correspond to the different states $PreState_1, \ldots, PreState_n$, and each state would have a transition to its appropriate next state $PostState_1, \ldots, PostState_n$. On the other hand, Contractor's model would have a single state corresponding to all the pre-states with nondeterministic transitions to the post-states, thus resulting in a less precise model.

Phase II: Refining the Invariant-based MTS

TEMI uses observed trace information to refine the invariant-based MTS by promoting to required those maybe transitions that correspond to observed invocations. REFINEINVARIANTMTS in Figure 9 describes this refinement algorithm. In [30], we summarize additional optimizations introduced to improve the scalability of TEMI as well as extensions that add further information to the inferred models. We omit their discussion for brevity as they are not part of the core algorithm.

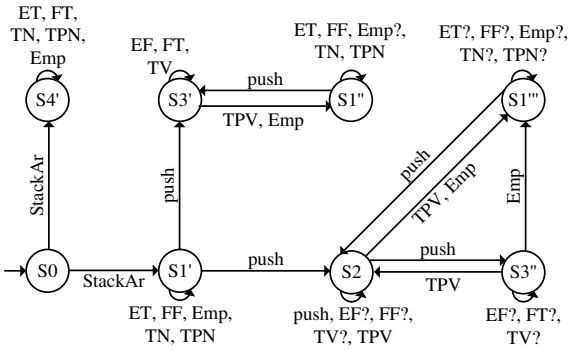

Figure 10: The `StackAr` MTS refined with invocation traces.

A direct approach to incorporating trace information into the invariant-based MTS is to simulate the traces on the MTS and promote each traversed maybe transition (denoted with '?' on the label) to a required transition (without '?'). However, this approach can result in imprecisions because states may be visited multiple times, and the produced model would not distinguish between the different visits. This can "stitch" together a required transition from one trace to a required transition from another trace, resulting in an invocation ordering that never occurred. For example, consider the direct refinement of the invariant-based MTS from Figure 8 based on the `StackAr` traces from Figure 3. The resulting MTS allows a spurious sequence in which — based on Trace 2 — `push(x)` from an empty stack (state S_1) to a full stack (S_3) is followed — based on Trace 3 — by `topAndPop()` to a partially full stack (S_2).

To avoid such issues, REFINEINVARIANTMTS enhances the direct refinement strategy by also refining the visited states (the refinement process is captured in lines 2–6). When REFINEINVARIANTMTS visits a state in the invariant-based MTS, it first splits it into two states (line 5, where REFINESTATE, shown in the bottom portion of Figure 9, is called). The first refined state (*nextSt'* in REFINESTATE) has only one incoming transition, which corresponds to the currently processed trace invocation (*currentEv*). The second refined state (*nextSt''*) keeps the remaining incoming transitions of the original state. Each state keeps all of the original outgoing transitions and self-transitions (lines 5–7 in REFINESTATE). The state splitting enables each state to express different behavior according to the incoming transitions. For example, `StackAr`'s MTS depicted in Figure 10 is obtained by refining the invariant-based MTS from Figure 8 with the traces from Figure 3. The `push(x)` invocation from S_5 to S_6 in Trace 2 (Figure 3) splits `StackAr`'s invariant-based state S_3 (Figure 8) into two new states S'_3 and S''_3 (Figure 10). S'_3 is reachable from empty-stack states, while S''_3 is reachable from a partially full stack. Furthermore, the transition $S'_1 \xrightarrow{push}_r S'_3$ is promoted to required since it has been observed in the traces.

Once the MTS is refined according to the traces, REFINEINVARIANTMTS uses the newly incorporated required transitions to further improve the model. First, REFINEINVARIANTMTS removes spurious nondeterministic transitions for each method, using a heuristic that if a state has non-deterministic transitions on a method, and some of those transitions are observed (required) while others are unobserved (maybe), then the unobserved transitions can be removed (lines 7–9) because they are likely a product of overly general invariants. For example, `StackAr`'s refined MTS in Figure 10 does not have a maybe transition from S'_3 on `topAndPop()` to a partially full stack state S_2. This is because such behavior was never observed. By contrast, a direct transition to an empty stack ($S'_3 \xrightarrow{topAndPop}_r S''_1$) was observed in Trace 2. This distinction between the refined states S'_3 and S''_3 was not present in the original invariant-based MTS due to `topAndPop()`'s incomplete postcondition.

Application	Type	Exec.	Libraries
StackArTester [15]	unit test	unit test	`StackAr`
jEdit [26]	text editor	end-user	`StringTokenizer`
jlGUI [28]	media player	end-user	`StringTokenizer`
Columba [9]	e-mail client	end-user	`Signature`, `Socket`, `SMTPProtocol`
jFTP [27]	file transfer	end-user	`Signature`, `SftpConnection`
JarInstaller [25]	packaging	end-user	`ZipOutputStream`
DaCapo Xalan [12]	benchmark	performance test	`NumberFormatStringTokenizer`, `ToHTMLStream`
Voldemort [55]	distributed database	unit tests	`Socket`

Figure 11: Eight applications that exercise the evaluated libraries.

4. EVALUATION

To evaluate the four model inference strategies, we used the four respective algorithms to infer models of nine open-source libraries. To generate the execution information necessary for our analysis, we exercised the libraries using other readily-available, open-source applications we found on the web. We then compared the quality of the models inferred by each of the four algorithms. Section 4.1 describes the applications and the libraries our evaluation uses. Section 4.2 explains the setup and goals of our evaluation. Sections 4.3, 4.4, and 4.5 present the evaluation results. Finally, Section 4.6 addresses the threats to the validity of our findings.

4.1 Subject Libraries and Applications

Our nine evaluation libraries span five categories:

1. Data structures (`DataStructures.StackAr`)
2–3. Data processing (`org.apache.xalan.templates.ElemNumber.NumberFormatStringTokenizer`, to which we will refer as `NFST`, and `java.util.StringTokenizer`)
4. Authentication and data-integrity verification (`java.security.Signature`)
5–6. Data streaming (`java.util.zip.ZipOutputStream` and `org.apache.xml.serializer.ToHTMLStream`)
7–9. Distributed communication and message exchange (`org.columba.ristretto.smtp.SMTPProtocol`, `net.sf.jftp.net.wrappers.SftpConnection`, and `java.net.Socket`)

To collect invocation traces for these libraries, we used eight open-source applications. Figure 11 describes the applications and the way we ran them to exercise the libraries. For example, we exercised `StringTokenizer`'s functionality by running jEdit as an end-user who edits and saves text, and `Socket`'s functionality by running Voldemort's unit tests that involve socket-based communication. We provided all techniques with the same set of traces and inferred invariants as inputs. The collected traces contained all invocations of the selected classes.

4.2 Evaluation Setup

This section describes our quality metrics, the process for assessing a model's quality based on ground-truth models, our hypotheses,

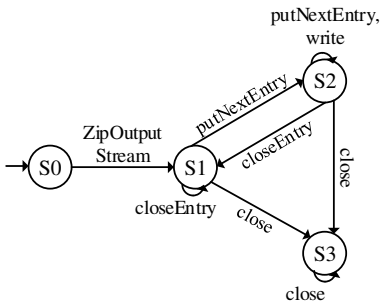

Figure 12: `ZipOutputStream`'s ground-truth model.

and our experiments.

4.2.1 Metrics

To measure the quality of a model, we compare it to a ground-truth model (see Section 4.2.2). We perform this comparison in terms of precision and recall [36]. Precision measures the fraction of one thousand traces generated by the inferred model that are allowed by the ground-truth. Recall measures the fraction of one thousand ground-truth traces that are allowed by the inferred model, suggesting how complete the model is. Since the evaluated models can have infinite traces, we restricted the length of the traces to twice the number of transitions in the ground-truth model.

4.2.2 Ground-Truth Models

Evaluating the quality of the inferred models requires a set of ground-truth models that represent the libraries' legal behavior. To this end, we used the ground-truth models that were manually extracted as a part of related work [16, 47]. We modified those models when we discovered imprecisions that we were able to validate by inspecting the source code, and when the models included non-public methods. We removed from the ground-truth models all transitions on methods that were never invoked in the collected traces. (This simplification equally impacts the recall of our techniques and of the existing techniques, and thus does not affect our conclusions. Meanwhile, the precision of the techniques is unaffected as the removed methods are not present in the inferred models.) For libraries without an existing ground-truth (`NFST`, `ToHTMLStream`, and `SftpConnection`), two doctoral students inspected the source code and API documentation, and manually constructed the models. As an example, Figure 12 depicts the ground-truth models for `ZipOutputStream`.

4.2.3 Hypotheses

Our evaluation tests the following hypotheses:

Hypothesis 1: Models inferred using invariants-only and trace-enhanced-invariants strategies are of significantly higher quality as compared to models inferred using traces-only and invariant-enhanced-traces strategies.

Hypothesis 2: Execution traces can circumvent imprecisions stemming from invariant inference that may be incomplete.

Hypothesis 3: Invariant filtering is a necessary aspect of invariants-based dynamic model inference techniques.

4.2.4 Experimental Design

For each library, we inferred seven models: traditional k-tail and SEKT with $k \in \{1, 2\}$, optimistic and pessimistic TEMI, and CONTRACTOR++. The pessimistic TEMI model removes *maybe* transitions (thus treating unobserved invocations as illegal), while the optimistic model retains all transitions. We do not report results of k-tail for $k > 2$ because those k values led to fewer merges, which lowered the recall without notable precision improvement.

In the experiments, for each library, we performed three steps: (1) execute applications that use the library to generate traces, (2) run the seven inference algorithms on those traces, and (3) measure the precision and recall of the inferred models. The implementation, observed traces, inferred models, and ground-truth models are publicly available: `http://softarch.usc.edu/wiki/doku.php?id=inference:start`.

4.3 Inferred Model Precision and Recall

This section assesses the quality of the inferred models, as compared to the ground truth. Figure 13 shows the precision (**P**) and recall (**R**) of each algorithm's models. Figure 13 distinguishes the algorithms along two dimensions: The algorithms developed fully as part of our research, placed in the central portion of the table, are separated from the existing algorithms by the two vertical lines; the different shadings distinguish implementations of traces-only and invariant-enhanced-traces strategies from the algorithms that implement invariants-only and trace-enhanced-invariants strategies. Next, we (1) compare the models' precision and recall; (2) analyze the reasons for better recall of models inferred via the invariants-only and trace-enhanced-invariants strategies; and (3) explain the differences between models inferred using those two strategies.

For all libraries, except `StringTokenizer`, CONTRACTOR++ and TEMI produce models of superior recall and comparable or better precision than the traditional k-tail and SEKT algorithms. For example, the k-tail models of `SMTPProtocol` allow only a single message to be sent before terminating the connection. The TEMI model correctly generalizes the observed behavior and allows multiple messages, which improves the recall. Compared with SEKT, for example, the precision of the optimistic TEMI is lower in the cases where TEMI either included an erroneous transition or inferred erroneous states due to incomplete invariants. Pessimistic TEMI does resolve the imprecisions due to erroneous transitions by considering only required transitions.

Figure 13 indicates that TEMI and CONTRACTOR++, have significantly higher recall than the k-tail-based algorithms. This is because the invariants help to distinguish between those invocations that change program state, in turn restricting future invocations, and those that do not. For example, `ToHTMLStream`'s invariants indicate no restrictions on its method invocations and the CONTRACTOR++ model has a single state with a self-transition for every method. In contrast, the k-tail models capture irrelevant invocation restrictions inferred from the traces. Furthermore, the results confirm that, while the traces-only strategy has high precision on average, its complete dependence on execution traces can result in precision-lowering spurious merges (e.g., this happened for models of `StackAr` and `SftpConnection`).

Our evaluation results also suggest that TEMI is slightly-to-moderately more precise than CONTRACTOR++, while having nearly-identical recall. The differences in precision stem from the ways these two approaches construct model states and the way TEMI incorporates the observed invocations. As discussed in Section 3.4, multiple TEMI states may be logically mapped to a single CONTRACTOR++ state; the CONTRACTOR++ state may have additional, precision-lowering, transitions that do not exist in the TEMI states.

The difference in precision is especially pronounced for `StringTokenizer` and `SMTPProtocol`, whose recall CONTRACTOR++ improved by up to 1%, but at a precision cost of 7–8%, as compared to optimistic TEMI. CONTRACTOR++'s model of `StringTokenizer` allowed illegal invocation sequences in which a first invocation of `hasMoreTokens()` returned *false* but a subsequent invocation returned *true*. This occurred because CONTRACTOR++ is not always able to precisely capture postconditions that relate post-state

Library	Traditional 1-tail		Traditional 2-tail		SEKT 1-tail		SEKT 2-tail		Optimistic TEMI		Pessimistic TEMI		CONTRACTOR++	
	P	R	P	R	P	R	P	R	P	R	P	R	P	R
StackAr	64%	30%	83%	30%	97%	30%	97%	30%	99%	94%	100%	71%	99%	94%
NFST	100%	21%	100%	21%	100%	21%	100%	21%	96%	57%	96%	44%	96%	57%
StringTokenizer	100%	52%	100%	51%	100%	51%	100%	50%	100%	51%	100%	50%	93%	52%
Signature	100%	61%	100%	61%	100%	61%	100%	61%	100%	88%	100%	88%	100%	88%
ToHTMLStream	100%	26%	100%	26%	100%	26%	100%	26%	100%	100%	100%	100%	100%	100%
ZipOutputStream	100%	37%	100%	36%	100%	37%	100%	35%	100%	63%	100%	47%	100%	63%
SMTPProtocol	100%	20%	100%	20%	100%	20%	100%	20%	96%	77%	100%	66%	88%	78%
Socket	94%	24%	97%	21%	100%	23%	100%	21%	100%	67%	100%	51%	100%	67%
SftpConnection	61%	31%	96%	29%	100%	30%	100%	29%	97%	48%	100%	33%	NA	NA
Average	95%	34%	98%	33%	100%	34%	100%	33%	99%	75%	99%	65%	97%	75%

Figure 13: Precision (P) and recall (R) comparison of k-tail, SEKT, TEMI, and the CONTRACTOR++ (enhanced Contractor) algorithms. The *Average* row excludes results for `SftpConnection`, because CONTRACTOR++ ran out of memory during model generation.

values to pre-state values. For `SMTPProtocol`, TEMI removed unobserved transitions on `getState()`, which remained in the CONTRACTOR++ model due to incomplete invariants.

In general, the causes behind CONTRACTOR++'s imprecision were more varied than those that impacted the recall of TEMI: While TEMI's recall may deteriorate because of overly restrictive invariants abstracted by CONTRACTOR++, CONTRACTOR++'s precision may be hampered by incomplete invariants, intricate relationships between invariants, and invocation dependencies that invariants cannot capture. We note that, while issues such as overly restrictive or incomplete invariants can be mitigated by collecting additional executions, invariant complexity and implied invocation dependencies cannot be mitigated in such a way.

> Our results strongly support Hypothesis 1 from Section 4.2: Utilizing program state information from invariants to create an initial model, before potentially augmenting it with information about invocation sequences from the execution traces, significantly improves the quality of dynamically inferred models. Our results also suggest that combining program state information with information about invocation sequences from execution traces results in near perfectly-precise models, as was the case for our invariant-enhanced-traces (SEKT) and trace-enhanced-invariants (TEMI) algorithms.

4.4 Sensitivity to Invariant Quality

In the real world, the collected execution data may be partial, and the results of invariant inference noisy. Model inference's aim is to maintain high precision under noise: Even a moderate drop in precision could render the produced model useless [48].

To evaluate the impact of invariant noise, we removed random subsets of invariant clauses and measured the resulting precision and recall of the TEMI and CONTRACTOR++ models. For each library, we generated 20 models for each of the cases when 10% and 20% of the invariant clauses are removed. The results in Figure 14 suggest that (1) the precision of pessimistic TEMI is robust to variations in invariant quality, (2) optimistic TEMI models outperform CONTRACTOR++ by yielding higher precision and recall, (3) noisy environments require enhancing invariants with trace information, and (4) decreased invariant quality negatively affects CONTRACTOR++'s performance. We elaborate on each point next.

1. Invariant incompleteness did not affect the near-perfectly precise (99%) pessimistic TEMI models. This confirms that the MTS refinement procedure (recall Section 3.4) appropriately introduces the trace information even when the initial model is imperfect. In particular, pessimistic TEMI models have 8–13% higher precision than optimistic TEMI and CONTRACTOR++ models, while also having 33% higher recall than comparably precise k-tail models, which are not affected by noisy invariants (see Figure 13).

2. When faced with noisy invariants, the optimistic TEMI models outperform CONTRACTOR++ in the average case by 1–4% in both precision and recall. The reasons were twofold: (1) Incomplete invariants add erroneous nondeterministic transitions; the degree of nondeterminism is higher in CONTRACTOR++ models, leading to lower precision. (2) Incomplete invariants cause TEMI's refinement procedure to correctly split states and subsequently remove the undesired maybe transitions only from the appropriate state.

3. In the average case, noisy invariants caused the TEMI models' precision to drop from 99% (Figure 13) to 89–91% (Figure 14); the precision dropped by 42% and 33% in the two extreme cases — `StackAr` and `Socket`. A reason for the extremes is that the states in TEMI's `Socket` model can have a number of incorrect nondeterministic transitions that confirm the connection (`isConnected()=true`) before it has been established. This highlights the crucial role of augmenting invariants-only models with trace information: `isConnected()` never returns this result in the actual traces and the erroneous transitions do not exist in the pessimistic TEMI models.

4. Incomplete invariants affect performance, making CONTRACTOR++ less efficient due to a higher number of allowed program states. This, in turn, results in a higher number of SMT queries that cause the scalability problems further discussed in Section 4.5.

> We conclude that invocation trace information is necessary to circumvent potential imprecisions in the invariants-only models. Hypothesis 2 holds for trace-enhanced-invariants models, as less reliable inferred program invariant information is augmented with information about invocation sequences from the execution traces. While lower quality invariants reduce the precision of the maybe transitions, the pessimistic TEMI models remained almost perfectly precise with unchanged recall, which makes pessimistic TEMI the most appropriate choice when an engineer is running an unfamiliar system to infer a model of a poorly documented library.

4.5 Impact of Invariant Filtering

As noted earlier, the high quality of invariants-only models is critically dependent on having meaningful and manageable invariants as inputs. This is best described using the differences between the original Contractor algorithm [18] and CONTRACTOR++, our enhancement of Contractor with invariant filters (recall Section 3.2).

Figure 15 outlines the model quality when the enhancements are **not** applied. Compared to CONTRACTOR++, when the input invari-

Library	Optimistic TEMI		Optimistic TEMI		Pessimistic TEMI		CONTRACTOR++		Optimistic TEMI		Pessimistic TEMI		CONTRACTOR++	
	full invariant set		10% invariants removed						20% invariants removed					
	P	R	P	R	P	R	P	R	P	R	P	R	P	R
StackAr	99%	94%	67%	96%	100%	71%	70%	95%	66%	96%	100%	80%	61%	97%
NFST	96%	57%	90%	59%	96%	44%	86%	60%	86%	60%	96%	44%	82%	69%
StringTokenizer	100%	51%	90%	77%	100%	50%	92%	69%	82%	79%	100%	50%	91%	72%
Signature	100%	88%	96%	90%	100%	88%	96%	88%	92%	91%	100%	88%	88%	77%
ToHTMLStream	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
ZipOutputStream	100%	63%	100%	69%	100%	47%	100%	69%	100%	76%	100%	47%	100%	73%
SMTPProtocol	96%	77%	96%	77%	100%	66%	86%	74%	94%	78%	100%	66%	80%	62%
Socket	100%	67%	61%	69%	100%	51%	NA	NA	58%	71%	100%	51%	NA	NA
Average	99%	76%	91%	81%	99%	67%	90%	79%	89%	83%	99%	67%	86%	79%

Figure 14: Precision (P) and recall (R) comparison of TEMI, and the CONTRACTOR++ algorithms in a noisy environment (when some of the invariants are removed). TEMI is more robust to the noise, with almost-perfectly precise pessimistic TEMI, while delivering slightly higher recall on average. The *Average* excludes results for `Socket`, because CONTRACTOR++ ran out of memory during model generation.

ants are not filtered (**No Invariant Filtering** in Figure 15), a modest 1% average increase in precision comes at the expense of a sizable drop in recall (up to 54%, in the case of `SMTPProtocol`). When the different method return points are not handled separately (**No Method Distinction**), the resulting models are 25% less precise on average, although the average recall increases by 10%. Omitting our enhancements also accentuated Contractor's scalability problems, denoted as *NA* values, discussed next.

Each CONTRACTOR++ SMT query includes the invariants of all methods, and such a query is generated for every possible combination of methods' invariant evaluations (recall Section 3.2). Hence, CONTRACTOR++ queries are longer and more resource consuming than TEMI's queries. In the case of `SftpConnection`, which has 22 methods with 684 invariant clauses, the SMT solver runs out of memory (Figure 13); this happens in five additional cases when our enhancements are not applied (Figure 15), as well as the one case with noisy invariants (Figure 14). Since CONTRACTOR++ is built using the combination of Python and C, the memory is allocated by the operating system, which kills the SMT solver's process once the memory consumption makes the system unstable.

Overall, our evaluation results confirm Hypothesis 3, as invariant filters that keep only a limited set of relevant invariant types have shown to be crucial to enhancing the scalability as well as the quality of techniques that implement invariants-only and trace-enhanced-invariants strategies.

4.6 Threats to Validity

We now outline the threats to our evaluation's validity and discuss our mitigation strategies.

Library	CONTRACTOR++		No Invariant Filtering		No Method Distinction	
	P	R	P	R	P	R
StackAr	99%	94%	99%	94%	77%	100%
NFST	96%	57%	98%	40%	78%	66%
StringTokenizer	93%	52%	91%	77%	47%	77%
Signature	100%	88%	100%	78%	NA	NA
ToHTMLStream	100%	100%	100%	100%	NA	NA
ZipOutputStream	100%	63%	NA	NA	NA	NA
SMTPProtocol	88%	78%	100%	24%	NA	NA
Socket	100%	67%	96%	60%	86%	68%

Figure 15: Precision (P) and recall (R) of Contractor models with and without SEKT- and TEMI-specific invariant filters.

Ground-truth bias. The ground-truth models were, in part, manually constructed. This may make them biased to the specifier's expertise. To mitigate this threat, we used the publicly available ground-truths from other researchers [16, 47], and modified them only if we were able to validate the modifications by inspecting the source code. We also made multiple iterations over the resulting models with two specifiers.

Subject libraries and applications. The selection of libraries and applications that invoke them may bias the evaluation results. For instance, models of a well-known library may not be representative of dynamically inferred models in general. Similarly, mature applications may be more careful in using a library's functionality. To this end, we selected libraries of different types and popularity (e.g., well documented Java libraries vs. less widely known `NFST`). We also used applications from several different domains and of different maturities, popularities, and sizes (e.g., widely used Voldemort vs. less popular JarInstaller). Finally, we had to address the bias stemming from inferring models based on traces obtained from unit and integration tests. Hence, we collected some of the utilized traces by executing open-source applications in ways that an end-user would use them to explore the available features.

Metrics. There are multiple ways of comparing the generated models with the corresponding ground-truth models (e.g., recall and precision of the simulated traces vs. graph comparison), which could potentially yield different results. Our mitigation strategy used metrics that have been proposed and adopted by the research community [36], and are consistent with our goal of comparing model behavior, not structural similarity.

5. OUR FINDINGS' IMPACT

While studying inferred model quality is an interesting exercise on its own, our findings, which confirm the high quality and robustness of trace-enhanced-invariants models, should motivate further application of model inference in practice. Below, we briefly elaborate on the utility of having the different types of generated models, and discuss the potential impact of our higher-quality trace-enhanced-invariants models on several development activities.

Model quality. The high overall quality of the inferred models can aid a variety of software development tasks including API understanding [1, 18], debugging, test generation [41], and runtime fault detection [22, 48]. For example, during debugging a programmer can analyze if a given library is invoked as expected. Similarly, an inferred trace-enhanced-invariants model that exhibits high recall (i.e., that is complete) can help to detect invocations that violate the library's protocol [48], while avoiding spurious warnings.

Required vs. maybe transitions. The TEMI models contain observed required transitions and unobserved maybe transitions. This dichotomy is useful for several reasons. For example, as discussed in Section 4.4, when the available executions are not comprehensive, the resulting invariants can be noisy. In such cases, a developer can rely on TEMI models due to almost perfect precision of the required transitions. In addition, the required MTS transitions can be augmented with frequencies of method invocations. This can benefit recommender systems [6, 7, 42, 50] in prioritizing examples by selecting those that commonly occur in actual executions.

Program state information. While our evaluation focused on the invocation sequences, our models also relate the model states to the internal program states. Hence, instead of having to gain a comprehensive understanding of a library's source code or to strictly rely on the available API-level documentation, a programmer can use the state information to "peek inside" the library's implementation. Similarly, tools that detect protocol violations (e.g., [48]) may provide more informative warnings by referring to program state.

Trace-oriented models. The invariant-enhanced-traces models can be more appropriate for development tasks that require trace-specific information, despite their significantly lower recall than that of the TEMI models. For example, detailed analysis of how an unfamiliar program communicates with a library requires compact, yet accurate representation of the traces themselves as opposed to a representation of the library's full API protocol. The SEKT algorithm produces such models, while the k-tail algorithm is less reliable due to its sole dependence on execution traces.

6. RELATED WORK

Program invariants have been used to directly synthesize FSM models [18], and to augment the k-tail algorithm with transition invariants [41]. As noted in Sections 4.3 and 4.4, TEMI is more appropriate than CONTRACTOR++ for dynamic specification mining because (1) it has higher precision, (2) it is more resilient to invariant noise, and (3) unlike Contractor, it distinguishes between observed and unobserved invocations. By contrast, starting from the observed executions and using Daikon [20] to infer transition invariants [41] results in other imprecisions, as discussed in Section 3.3.

The k-tail algorithm [5] serves as a basis for many FSM-inference techniques from invocation traces [10, 36, 37, 39, 41, 49, 56]. These algorithms (1) extend k-tail to improve its precision or recall [10, 39, 49, 56], (2) build larger frameworks with k-tail as the inference algorithm [37, 49], and (3) enhance the models with information about invocation probabilities [37] and program state and method parameters [41]. Additional merges can make the models more compact [10, 49], while stricter merging conditions based on pairwise sequencing invariants can improve precision [39, 56]. In general, these approaches' recall is only as good as the traditional 1-tail and, for our evaluation libraries, their precision cannot surpass the precision of SEKT. Synoptic [4], CSight [3], and Perfume [43, 44] use the CEGAR [8] approach to create a coarse initial model, and then refine it using counterexamples that falsify temporal invariants. InvariMint [2] presents a declarative specification language for expressing model-inference algorithms, and improves the efficiency of algorithms, but neither their precision nor recall.

There are other ways to aid development tasks than inferring models, such as detecting method invocation patterns [21, 60] and inter-object sequence charts [32, 38]. These patterns can detect code anomalies [22, 48]. Scenario-based views of a system's behavior, such as parametrized [38] and symbolic [32] sequence charts, facilitate understanding of a system's runtime interactions.

While we used both CONTRACTOR++ and TEMI in combination with invariants inferred by Daikon [20], invariants-only and trace-enhanced-invariants techniques can be adapted to work with other invariant-inference techniques. Examples of such techniques include techniques that combine inference with symbolic program execution [11] and enhance it by using simple, manually written initial invariants [57] or manually-written relations between methods [35].

Static [19, 53] and hybrid techniques [17, 58] provide two alternatives to specification mining based strictly on invocation traces. Shoham et al. [53] infer, from client-side code, FSM models that over-approximate the actual invocation sequences. By contrast, our algorithms work on traces generated from multiple parts of the code and, potentially, from multiple applications. De Caso et al. [19] statically analyze C programs for invariants and use Contractor to create models that allow more behavior than the ground truth.

ADABU [17] infers the concrete program state by statically finding side-effect-free invocations and combines that information with test case executions. While the concrete program state is also abstracted using predicates, these predicates are predetermined and do not relate multiple variables (e.g., ADABU abstracts integers only as negative, zero, or positive). Together with test case generation, ADABU can improve model quality, enabling code verification [16]. However, this requires tailored unit test executions. Compared to TEMI, ADABU uses limited invariants and infers one model per runtime object, which hampers its applicability to rich classes and executions that involve many objects of the same type. Whaley et al. [58] create a separate submodel for each field of a class, analyzing if a method modifies a given field, and creating a 1-tail model that combines static and dynamic information about method invocations with respect to that field. Unlike TEMI, this approach requires static analysis, creates multiple submodels, and considers only one-step history (1-tail), limiting its applicability.

7. CONCLUSIONS

Using a software library is a non-trivial task hampered by a lack of appropriate documentation for describing the required but often-implicit invocation protocols. This paper studied how different model-inference strategies perform when applied to libraries whose behavior is exercised using real software. The recent scalability improvements of the dynamic inference techniques have resolved many of the obstacles to their application in the real world: these techniques are now able to handle large sets of execution traces [4, 34] and large sets of runtime data values [45, 61]. However, there are still noticeable gaps in understanding the quality of the models produced by model-inference techniques.

As part of this research, we enhanced one existing technique, and presented two novel algorithms, SEKT and TEMI, that combine execution traces with automatically inferred program-state invariants. Our evaluation demonstrates that invariants-only and trace-enhanced-invariants significantly outperform the traces-only and invariant-enhanced-traces strategies. Trace-enhanced-invariants models, produced by TEMI, also exhibit superior recall, while being robust to noisy inputs. Our results highlight the significant impact of using program-state information to infer high-quality models. In addition, our research highlights the benefits of combining different types of runtime information to enhance inferred models and, in turn, to effectively support development tasks. In our future work, we plan to study whether this combination can enhance model-inference for concurrent libraries and multi-object protocols [34].

8. ACKNOWLEDGMENTS

This work has been supported by the National Science Foundation under award numbers 1117593, 1218115, and 1321141. The work has also been supported in part by Infosys Technologies Ltd.

9. REFERENCES

[1] N. Beckman, D. Kim, and J. Aldrich. An empirical study of object protocols in the wild. In *the European Conference on Object-Oriented Programming (ECOOP)*, 2011.

[2] I. Beschastnikh, Y. Brun, J. Abrahamson, M. D. Ernst, and A. Krishnamurthy. Unifying FSM-inference algorithms through declarative specification. In *the International Conference on Software Engineering (ICSE)*, 2013.

[3] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy. Inferring Models of Concurrent Systems from Logs of their Behavior with CSight. In *the International Conference on Software Engineering (ICSE)*, 2014.

[4] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *the Joint Meeting of European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*, 2011.

[5] A. Biermann and J. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers*, 21(6), 1972.

[6] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *the Joint Meeting of European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*, 2009.

[7] R. P. Buse and W. Weimer. Synthesizing API usage examples. In *the International Conference on Software Engineering (ICSE)*, 2012.

[8] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided Abstraction Refinement. In *Computer Aided Verification*, pages 154–169, 2000.

[9] Columba e-mail client. `http://sourceforge.net/projects/columba`, 2013.

[10] J. Cook and A. Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology*, 7(3), 1998.

[11] C. Csallner, N. Tillmann, and Y. Smaragdakis. DySy: Dynamic symbolic execution for invariant inference. In *the International Conference on Software Engineering (ICSE)*, 2008.

[12] DaCapo benchmark. `http://www.dacapobench.org`, 2009.

[13] B. Dagenais and M. Robillard. Creating and evolving developer documentation: understanding the decisions of open source contributors. In *the Symposium on Foundations of Software Engineering (FSE)*, 2010.

[14] B. Dagenais and M. Robillard. Recovering traceability links between an API and its learning resources. In *the International Conference on Software Engineering (ICSE)*, 2012.

[15] The Daikon invariant detector. `http://groups.csail.mit.edu/pag/daikon`, 2009.

[16] V. Dallmeier, N. Knopp, C. Mallon, G. Fraser, S. Hack, and A. Zeller. Automatically generating test cases for specification mining. *IEEE Transactions on Software Engineering*, 38(2), 2012.

[17] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with ADABU. In *the Workshop on Dynamic Analysis (WODA)*, 2006.

[18] G. de Caso, V. Braberman, D. Garbervetsky, and S. Uchitel. Automated abstractions for contract validation. *IEEE Transactions on Software Engineering*, 38(1), 2012.

[19] G. de Caso, V. Braberman, D. Garbervetsky, and S. Uchitel. Enabledness-based program abstractions for behavior validation. *ACM Transactions on Software Engineering and Methodology*, 22(3), 2013.

[20] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1), 2007.

[21] M. Gabel and Z. Su. Javert: Fully automatic mining of general temporal properties from dynamic traces. In *the Symposium on Foundations of Software Engineering (FSE)*, 2008.

[22] M. Gabel and Z. Su. Online inference and enforcement of temporal properties. In *the International Conference on Software Engineering (ICSE)*, 2010.

[23] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is still so hard. *IEEE Software*, 26(4), 2009.

[24] C. Ghezzi, M. Pezzè, M. Sama, and G. Tamburrelli. Mining Behavior Models from User-intensive Web Applications. In *the International Conference on Software Engineering (ICSE)*, 2014.

[25] JarInstaller. `http://sourceforge.net/projects/kurumix`, 2013.

[26] jEdit. `http://www.jedit.org`, 2014.

[27] JFtp client. `http://j-ftp.sourceforge.net`, 2013.

[28] jlGUI. `http://www.javazoom.net/jlgui/jlgui.html`, 2010.

[29] I. Krka, Y. Brun, G. Edwards, and N. Medvidovic. Synthesizing partial component-level behavior models from system specifications. In *the Joint Meeting of European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*, 2009.

[30] I. Krka, Y. Brun, and N. Medvidovic. Automatically mining specifications from invocation traces and method invariants. Technical Report CSSE-2013-509, Center for Systems and Software Engineering, University of Southern California, 2013.

[31] I. Krka, Y. Brun, D. Popescu, J. Garcia, and N. Medvidovic. Using dynamic execution traces and program invariants to enhance behavioral model inference. In *the International Conference on Software Engineering New Ideas and Emerging Results Track (ICSE NIER)*, 2010.

[32] S. Kumar, S.-C. Khoo, A. Roychoudhury, and D. Lo. Inferring class level specifications for distributed systems. In *the International Conference on Software Engineering (ICSE)*, 2012.

[33] K. G. Larsen and B. Thomsen. A modal process logic. *Logic in Computer Science*, 1988.

[34] C. Lee, F. Chen, and G. Roşu. Mining parametric specifications. In *the International Conference on Software Engineering (ICSE)*, 2011.

[35] K. Li, C. Reichenbach, Y. Smaragdakis, and M. Young. Second-order constraints in dynamic invariant inference. In *the Joint Meeting of European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*, 2013.

[36] D. Lo and S. Khoo. QUARK: Empirical assessment of automaton-based specification miners. In *the Working Conference on Reverse Engineering (WCRE)*, 2006.

[37] D. Lo and S. Khoo. SMArTIC: Towards building an accurate, robust and scalable specification miner. In *the Symposium on Foundations of Software Engineering (FSE)*, 2006.

[38] D. Lo and S. Maoz. Scenario-based and value-based specification mining: Better together. In *the International Conference on Automated Software Engineering (ICSE)*, 2010.

[39] D. Lo, L. Mariani, and M. Pezzè. Automatic steering of behavioral model inference. In *the Joint Meeting of European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*, 2009.

[40] D. Lo, L. Mariani, and M. Santoro. Learning extended fsa from software: An empirical assessment. *Journal of Systems and Software*, 85(9), 2012.

[41] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *the International Conference on Software Engineering (ICSE)*, 2008.

[42] K. Muşlu, Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Speculative analysis of integrated development environment recommendations. In *the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2012.

[43] T. Ohmann, M. Herzberg, S. Fiss, A. Halbert, M. Palyart, I. Beschastnikh, and Y. Brun. Behavioral Resource-Aware Model Inference. In *International Conference On Automated Software Engineering (ASE)*, Västerås, Sweden, 2014.

[44] T. Ohmann, K. Thai, I. Beschastnikh, and Y. Brun. Mining Precise Performance-Aware Behavioral Models from Existing Instrumentation. In *the International Conference on Software Engineering New Ideas and Emerging Results (ICSE NIER) track*, 2014.

[45] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, et al. Automatically patching errors in deployed software. In *the Symposium on Operating Systems Principles (SOSP)*, 2009.

[46] N. Polikarpova, I. Ciupa, and B. Meyer. A comparative study of programmer-written and automatically inferred contracts. In *the International Symposium on Software Testing and Analysis (ISSTA)*, 2009.

[47] M. Pradel, P. Bichsel, and T. R. Gross. A framework for the evaluation of specification miners based on finite state machines. In *the International Conference on Software Maintenance (ICSM)*, 2010.

[48] M. Pradel and T. R. Gross. Leveraging test generation and specification mining for automated bug detection without false positives. In *the International Conference on Software Engineering (ICSE)*, 2012.

[49] S. P. Reiss and M. Renieris. Encoding program executions. In *the International Conference on Software Engineering (ICSE)*, 2001.

[50] R. Robbes and M. Lanza. How program history can improve code completion. In *the International Conference on Automated Software Engineering (ASE)*, 2008.

[51] M. Robillard. What makes APIs hard to learn? Answers from developers. *IEEE Software*, 26(6), 2009.

[52] M. Schur, A. Roth, and A. Zeller. Mining behavior models from enterprise web applications. In *the Joint Meeting of European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*, 2013.

[53] S. Shoham, E. Yahav, S. J. Fink, and M. Pistoia. Static Specification Mining Using Automata-Based Abstractions. *IEEE Transactions on Software Engineering*, 34(5), 2008.

[54] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons, 2009.

[55] Project Voldemort. `http://www.project-voldemort.com`, 2014.

[56] N. Walkinshaw and K. Bogdanov. Inferring finite-state models with temporal constraints. In *the International Conference on Automated Software Engineering (ASE)*, 2008.

[57] Y. Wei, C. A. Furia, N. Kazmin, and B. Meyer. Inferring better contracts. In *the International Conference on Software Engineering (ICSE)*, 2011.

[58] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *the International Symposium on Software Testing and Analysis (ISSTA)*, 2002.

[59] T. Xie et al. Data mining for software engineering. *Computer*, 42(8), 2009.

[60] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: Mining temporal API rules from imperfect traces. In *the International Conference on Software Engineering*, 2006.

[61] Yices SMT Solver. `http://yices.csl.sri.com`, 2009.

[62] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst. Combined static and dynamic automated test generation. In *the International Symposium on Software Testing and Analysis (ISSTA)*, 2011.