Reducing the Cost of Path Property Monitoring Through Sampling

Matthew B. Dwyer, Madeline Diep, and Sebastian Elbaum
University of Nebraska - Lincoln
{dwyer,mhardojo,elbaum}@cse.unl.edu

Abstract

Run-time monitoring can provide important insights about a program's behavior and, for simple properties, it can be done efficiently. Monitoring properties describing sequences of program states and events, however, can result in significant run-time overhead. In this paper we present a novel approach to reducing the cost of run-time monitoring of path properties. Properties are composed to form a single integrated property that is then systematically decomposed into a set of properties that encode necessary conditions for property violations. The resulting set of properties forms a lattice whose structure is exploited to select a sample of properties that can lower monitoring cost, while preserving violation detection power relative to the original properties. Preliminary studies for a widely used Java API reveal that our approach produces a rich, structured set of properties that enables control of monitoring overhead, while detecting more violations than alternative techniques[1].

I. Introduction

Mechanisms for monitoring and checking *state properties*, i.e., predicates at specific program locations, through assertions and method pre and post-conditions are becoming standard features in programming languages. Recently, researchers have started to explore support for monitoring programs for conformance with user-defined *path properties* (also referred to as temporal, sequencing, or typestate properties) [1], [2], [3], [4], [5]. These properties can be used to encode, for example, constraints on the proper sequencing of calls on an API; to call `read()` on a `java.nio.channels.SocketChannel` one must first call `connect()`. A key challenge in monitoring path properties is run-time overhead. Even highly optimized implementations of monitors for such properties can incur significant overhead, e.g., up to 150% [2], when confronted with large numbers of instances to be monitored for complex properties that involve frequently executed calls.

To address this challenge we first consider the literature on mitigating the overhead of run-time monitoring of *state properties*. It is possible, for example, to statically eliminate the monitoring of locations whose properties can be derived by monitoring other locations [6], [7]. When that is not sufficient, monitoring can be activated and deactivated at run-time based on the program state as it relates to the monitored property [8], [9], [10]. Further overhead reduction can be achieved through sampling, which consists of selecting a subset of the events to monitor, e.g., [11], [12], [13], [14].

Inspired by this work in monitoring *state properties*, researchers have recently adapted approaches to statically eliminate [2], [15] and to dynamically activate and deactivate [4] monitoring probes for path properties. These approaches can be classified as *sound for falsification* [16] – they only report errors when the program violates the monitored property.

Adapting sampling approaches to path properties to control monitoring overhead, however, is more difficult and has not been well studied. It is more difficult because of the intrinsic dependencies between the observable elements of a path property. For *state properties*, one can easily sample in space and time. Spatial sampling involves selecting from the set of program points to be monitored and sampling over time involves selecting which occurrence of a monitored program point is processed. Either way, if a state property monitor detects a violation, the program has definitely violated the state property. For path properties, either form of sampling can lead to reports of false violations. Consider a simple property that enforces the strict alternation of two calls, $open$ and $close$, encoded as the regular expression $(open; close)*$ and a program execution $open, close, open, close$. If spatial sampling chooses to not monitor the site of the first $open$ call, then the occurrence of the first $close$ is reported as a violation since $close$ cannot come first. If temporal sampling chooses to suppress

[1]This material is based in part upon work supported by the National Science Foundation under Awards CCF-0429149, CNS-0454203, CCF-0541263, and CNS-0720654. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation

monitoring of the second occurrence of the $close$ call then the program ends in an non-accepting state of the property monitor and a violation is reported.

Sampling is a powerful technique for controlling monitoring cost, but for sound path property monitoring a new approach is needed. In this paper, we propose a novel approach to sound path property monitoring that samples the *structure* of a path property. The approach has 3 main elements:
(1) In contrast to the prevalent trend in static analysis and verification which advocates the use of small, focused properties, we demonstrate the value of large complex properties. Specifically, we automatically compose collections of related properties to form a single integrated property and leverage the structure of that property to produce sampling opportunities to control monitoring overhead.
(2) We systematically decompose a property into a lattice of properties each defined over a different restricted set of monitored observables. These properties, which we call *sub-alphabet properties*, are less expensive to monitor since they observe less, and will not report false positives.
(3) We define strategies for exploiting the structure of the lattice to sample the set of decomposed properties. While they vary, each strategy operates by selecting subsets of properties to deploy in a given program run. As we show, this can lead to reductions in the overhead of run-time monitoring while retaining violation detection power.

Section III defines the lattice of sub-alphabet properties and presents a general approach to construct it from any given set of path properties. Section IV presents a set of strategies for sampling the lattice. In Section V, we present the promising results of a preliminary study applying the approach to monitor properties of three applications that use the complex real-world Java `Hibernate` API [17]. We discuss related work in Section VI and conclude in Section VII. The next section provides an overview of our approach and its application to a small example.

II. Overview

We present an overview of current approaches for monitoring path properties, challenges to scaling those approaches, and the concept and benefits of our approach by considering the problem of monitoring for correct usage of the `java.nio.channels.SocketChannel` API. `SocketChannels` support networked IO in Java. The standard use-case for an instance of a `SocketChannel` is to *open* and *connect* a channel, then perform a series of *read* and *write* operations on the channel, and finally to *close* the channel. There are, however, many other operations that can be performed on `SocketChannels`, for example, open and connect can be performed in a single step with a call to `open(A)` passing a socket address.

Figure 1 lists a collection of path properties extracted from the `SocketChannel` API documentation [18], later referred as original properties. The English phrases, on the left, paraphrase text in the class documentation using the terminology of the specification pattern system [19]. Regular expressions encoding these path properties are expressed in the Laser FSA package's syntax [20]. In this presentation, regular expressions are defined over names of the public methods in the `SocketChannel` and `Socket` classes; in general, one would use method signatures rather than names to treat overloading. In this syntax, most operators, e.g., `*`, `|`, `.`, `?`, have their standard meaning. In addition, `~[ ... ]` denotes the complement of a symbol set, which is the disjunction of all symbols not listed between the brackets, and `;` denotes concatenation.

Existing approaches to monitoring path properties [2], [3], [4] involve instrumenting the program to observe a relevant program action, e.g., a method call. A stream of observations is then used to evaluate the path properties encoded as finite-state automata (FSA) to detect violations on-the-fly during program execution. In this setting, there are two components of run-time cost: (1) the number of observations generated and (2) the cost of monitoring each path property for each allocated instance of the class, e.g., `SocketChannel`.

Consider an execution of an application that uses a single instance of `SocketChannel` in the typical sequence

$$open; connect; read^k; write^k; close$$

where a^k denotes k repetitions of a. The basic cost of monitoring this application for the 7 properties in Figure 1 is $(2k+3)(c_o+7c_m)$ where c_o is the cost of executing the instrumentation to observe a method call and c_m is the cost of updating a property FSA. Reducing monitoring costs for such properties has been the target of recent research: c_m can be reduced by the use of clever data structures and algorithms [2], [3], the $(2k+3)$ can be reduced by removing instrumentation that is guaranteed to not contribute to a path property violation [15], and, in certain cases, the k terms can be eliminated by dynamically eliminating instrumentation when cyclic behavior cannot change the FSA monitor's state [4]. Despite these advances, there still remain path properties for real-world programs and APIs that cannot be effectively optimized and they can incur overhead of greater than 150% [2].

A. Our Approach

One way to reduce monitoring overhead is to integrate small properties into larger and more comprehensive properties. Expecting developers to write such specifications is problematic, but they can be constructed through specification mining [21] or by directly composing sets of smaller related properties. We take the latter approach. Figure 2 is

<table>
<tr><td>(1)</td><td><code>open()</code> or <code>open(A)</code> before anything</td><td><code>(open | openA); .*</code></td></tr>
<tr><td>(2)</td><td><code>close()</code> after <code>open()</code> or <code>open(A)</code></td><td><code>~[open,openA]*; ((open|openA); ~[close]*; close; .*)?</code></td></tr>
<tr><td>(3)</td><td><code>connect()</code> or <code>open(A)</code> before <code>read()</code> or <code>write()</code></td><td><code>~[read,write]* | ~[read,write]*;(connect|openA);.*</code></td></tr>
<tr><td>(4)</td><td>no <code>read()</code> or <code>write()</code> after <code>close()</code></td><td><code>~[close]*; (close; ~[read,write]*)?</code></td></tr>
<tr><td>(5)</td><td><code>socket()</code> before <code>shutdownIn(Out)put()</code></td><td><code>~[shutIn,shutOut]* | ~[shutIn,shutOut]*;socket;.*</code></td></tr>
<tr><td>(6)</td><td>no <code>read()</code> after <code>shutdownInput()</code></td><td><code>~[shutIn]*; (shutIn; ~[read]*)?</code></td></tr>
<tr><td>(7)</td><td>no <code>write()</code> after <code>shutdownOutput()</code></td><td><code>~[shutOut]*; (shutOut; ~[write]*)?</code></td></tr>
</table>

Fig. 1. `SocketChannel` **properties as specification patterns and regular expressions**

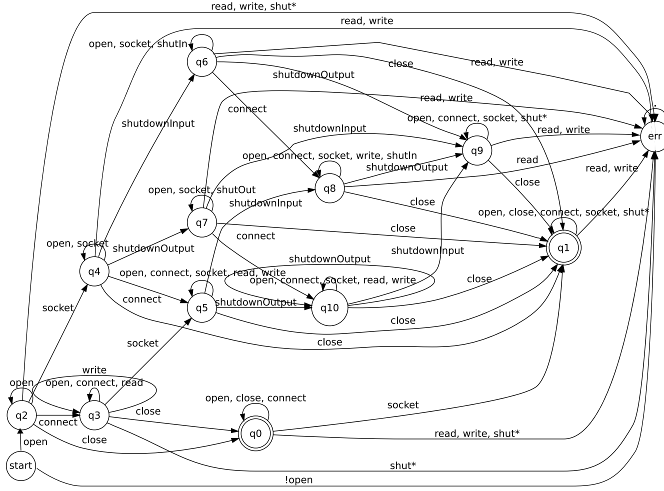

Fig. 2. Integrated Constraint FSA

the product automata constructed from the FSA for all 7 original properties in Figure 1; we denote this property ϕ. This *integrated path property* is less expensive to monitor than the 7 properties independently since only one FSA must be updated (it will only cost $(2k+3)(c_o+c_m)$), but it still detects the same violations as the original properties. The real value of the integrated path property, however, lies in defining a richer space of sub-languages that can be sampled to expose further cost-effectiveness tradeoffs in runtime monitoring.

For example, Figure 3 illustrates FSAs for alphabets $\{read, close\}$ (on the top) and $\{connect, read\}$ (in the middle). These properties were not in the original set of 7 properties, but can be generated by sampling the structure of ϕ[2]. Monitoring $\phi_{\{read,close\}}$ for the sequence of `SocketChannel` calls given above will cost $(1 + k)(c_o + c_m)$. Other sub-alphabet properties, for example $\{connect, read, close\}$, can encode a single property that enforces elements of multiple original properties, e.g., (3) and (4), yet avoids the cost of monitoring all symbols in those properties, e.g., `openA` or `write`. Property $\phi_{\{connect,read,close\}}$, shown at the bottom of Figure 3, can be monitored on the sample trace at a cost of $(2 + k)(c_o + c_m)$.

[2]Note that the automata in Figure 3 are limited by the original set of constraints extracted from informal API documentation. As such, they are incomplete and allow behavior that might normally be regarded as an error, e.g., $connect; connect$ is accepted by $\phi_{\{connect,read\}}$.

The penalty for the reduced monitoring cost of sub-alphabet properties is a potential loss of violation detection. For example $\phi_{\{read,close\}}$ would miss violations where the `SocketChannel` was not connected and $\phi_{\{connect,read\}}$ would miss violations where it was not closed. Note, however, that any violation of these two properties is guaranteed to be a violation of the integrated path property.

Compared with the original 7 properties, the space defined by ϕ includes 238 properties capable of violation detection (including those illustrated in Figure 3). This richer population offer properties that vary in monitoring cost and violation detection ability, yet, as a whole, are sound with respect to violation detection relative to the original properties. We can then strategically sample from this space to trade violation detection power and monitoring cost. For example, we might select a set of low-cost properties, e.g., by avoiding symbols that appear frequently like $read$, that together include a large number of different rejecting transition sequences.

In the next section, we define this property space and show that it forms a lattice ordered, intuitively, by cost and violation-detection effectiveness.

III. A Space of Necessary Properties

In this section, we describe foundational concepts and algorithms for generating a space of properties that can be sampled to monitor path properties during program executions. Properties in this space are automatically generated from a given set of target properties and (1) represent necessary conditions for the target properties to hold, (2) offer a diversity in cost and violation-detection power, and (3) are related to each other via a refinement relation. We begin with background on monitoring of path properties.

A. Monitoring Path Properties

Path properties can be expressed in a variety of formalisms, e.g., [19]. Regardless of the formalism, developers define properties in terms of *observations* of a program's behavior. In general, an observation may be defined in terms of a change in the data state of a program, the execution of a statement or class of statements, or some combination of the two. For simplicity, in our presentation we only consider observations that correspond to method

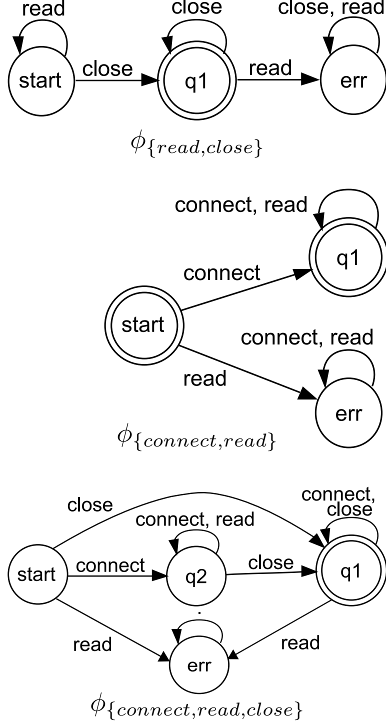

Fig. 3. Sub-alphabet FSA

calls. We define an *observable alphabet*, Σ, as a set of symbols that encode observations of program behavior.

For run-time monitoring, the most common form of path property specification used is a deterministic *finite state automaton* (FSA) [22]. An FSA is a tuple $\phi = (S, \Sigma, \delta, s_0, A)$ where: S is a set of states, Σ is the alphabet of symbols, $s_0 \in S$ is the initial state, $A \subseteq S$ are the accepting states and $\delta\colon S \times \Sigma \rightarrow S$ is the state transition function. We use $\Delta\colon S \times \Sigma^+ \rightarrow S$ to define the composite state transition for a sequence of symbols from Σ. We define a *trap* state as $s_{trap} \in S$ such that $\neg\exists\sigma \in \Sigma^* : \Delta(s_{trap}, \sigma) \in A$. A property defines a *language*, or set, of words $L(\phi) = \{\sigma \mid \sigma \in \Sigma^* \wedge \Delta(s_0, \sigma) \in A\}$.

FSA monitoring involves instrumenting a program to detect each occurrence of an observable, $a \in \Sigma$. The analysis stores the current state, $s_c \in S$, which is initially s_0, and at each occurrence of an observable it updates the state to $s_c = \delta(s_c, a)$ to track the progress of the FSA in recognizing the sequence of symbols for the program execution. The analysis detects a violation whenever $s_c = s_{trap}$ or when the program terminates with $s_c \notin A$. We say that a program execution, t, *violates* a property, ϕ, if the sequence of observable symbols, σ, corresponding to t ends in a non-accepting state, $\sigma \notin L(\phi)$.

Different approaches to checking program-property conformance produce qualitatively different results. We want analyses that are sound with respect to reporting violations; such an analysis may fail to report a violation when it occurs, but it will never report a violation when one does not occur. Moreover, we are interested in producing such reports while reducing checking overhead. Our basic strategy involves checking for violations of necessary conditions for ϕ; ϕ' is a necessary condition for ϕ if both are defined over the same alphabet and $L(\phi) \subseteq L(\phi')$.

B. Sub-alphabet Properties and the Lattice

To significantly reduce the cost of property monitoring, one must reduce the number of observable occurrences that are processed. One way to achieve this is to consider a subset of the observables in a property.

Definition 3.1 (Sub-alphabet Property): Given an FSA, $\phi = (S, \Sigma, \delta, s_0, A)$, a *sub-alphabet property* is $\phi_{\Sigma'} = (S, \Sigma', \delta', s_0, A)$, where $\Sigma' \subseteq \Sigma$ and

$$\forall a \in \Sigma' : \delta'(s, a) = \delta(s, a)$$
$$\forall a \in \Sigma - \Sigma' : \delta'(s, \epsilon) = \delta(s, a)$$

This definition yields sub-alphabet properties that are non-deterministic automata. For convenience, we use $\phi_{\Sigma'}$ to denote any equivalent automaton, e.g., a determinized minimized FSA accepting the same language.

Since the sub-alphabet properties for ϕ are defined over a different language, we cannot simply consider language containment to determine whether they constitute necessary conditions for ϕ. Instead, we require that every word over ϕ's alphabet whose projection (onto Σ') is rejected by $\phi_{\Sigma'}$ must be rejected by ϕ.

Proposition 3.1 (Necessary Sub-alphabet Properties): Let ϕ be an FSA, $\Sigma' \subseteq \Sigma$, and $\pi_{\Sigma'}\colon \Sigma^* \rightarrow \Sigma'^*$ be the projection of words over Σ onto words over Σ', then

$$\forall \sigma \in \Sigma^* : \pi_{\Sigma'}(\sigma) \notin L(\phi_{\Sigma'}) \Rightarrow \sigma \notin L(\phi)$$

By Definition 3.1, ϕ and $\phi_{\Sigma'}$ are isomorphic and every transition sequence in ϕ has a corresponding sequence in $\phi_{\Sigma'}$ where $a \in \Sigma - \Sigma'$ is replaced by ϵ. After a word σ, $\phi_{\Sigma'}$ can be in a set of states $\Delta'(s_0, \pi'_{\Sigma}(\sigma))$, since $\phi_{\Sigma'}$ is non-deterministic, that is guaranteed to be a superset of the state of ϕ after σ, i.e., $\Delta(s_0, \sigma)$. Consequently,

$$(\Delta'(s_0, \pi_{\Sigma'}(\sigma)) \cap A) = \emptyset \Rightarrow \Delta(s_0, \sigma) \notin A$$

Thus any sub-alphabet property is a necessary condition for the original property. We denote the sub-alphabet property relation $\phi_{\Sigma'} \preceq \phi_{\Sigma}$, for alphabets $\Sigma' \subseteq \Sigma$.

Given an FSA ϕ, we define a lattice of sound properties for violation reporting relative to ϕ by considering each of its sub-alphabet properties. The alphabet power-set lattice, $(\mathcal{P}(\Sigma), \subseteq)$, induces a lattice of sub-alphabet properties.

Definition 3.2 (Lattice of Sub-alphabet Properties): Given a property ϕ, the lattice of sub-alphabet properties, L_ϕ consists of the set $\{\phi_{\Sigma'} | \Sigma' \subseteq \Sigma \wedge L(\phi_{\Sigma'}) \neq \Sigma'^*\}$ ordered by $\preceq$. $\top = \phi$, since $\forall \phi' \in L_\phi : \phi' \preceq \phi$.

In general, there is a set of least elements $\bot = \{\phi_\bot | \neg\exists \phi' \in L_\phi - \{\phi_\bot\} : \phi' \preceq \phi_\bot\}$. Meet is defined as $\phi_{\Sigma_1} \sqcap \phi_{\Sigma_2} = \phi_{\Sigma_1 \cup \Sigma_2}$.
We note that this lattice explicitly excludes *trivial* properties that are incapable of rejecting a word, i.e., $L(\phi_{\Sigma'}) = \Sigma'^*$. Such properties lie toward the bottom of the lattice and the least properties are the ones with the smallest alphabets that are capable of rejecting a word. The sub-alphabet powerset and property lattices are isomorphic (except for trivial properties), so we use the alphabet to denote the corresponding property. In our presentation, we use Σ to denote the alphabet of the $\top$ property of a lattice L_ϕ.

This lattice can be constructed using well-known algorithms on automata [22]. Since the lattice has at most $2^{|\Sigma|}$ properties it can be costly to construct for large alphabets. Determinizing non-deterministic automata may incur an exponential blowup. However, none of the path properties we have analyzed have the pathological structure that leads to this blowup. In the next Section we discuss our lattice sampling strategy which permits several optimizations to control property sample construction cost.

IV. Sampling the Property Space

Multiple strategies can be applied to select properties from L_ϕ for monitoring. In this section, we present several approaches for selecting sets of properties to be monitored that exploit the structure of the lattice, the structure of individual properties in the lattice, and estimates of the cost of monitoring properties. We compose these selection functions to iteratively construct a sample of properties.

A. Selecting Properties

The subsumption property inherent in the lattice can be exploited to restrict sampling to properties with potential *added value* in terms of violation detection relative to a given property set.

Definition 4.1 (Added Violation-detection): Given a set of properties $P \subseteq L$,

$$addVd(P, L) = \{\phi' | \phi' \in L \wedge \neg\exists \phi'' \in P : \phi' \preceq \phi''\}$$

This defines the set of properties in lattice L that are not subsumed by properties in P and thus have the potential to detect additional violations.

When sampling a space one often wishes to achieve a measure of *diversity* in selecting sample properties. One notion of diversity for properties in L_ϕ measures the magnitude of differences in property alphabet sizes.

Definition 4.2 (Max Alphabet Difference): Given $P \subseteq L$ and alphabet $\alpha \subseteq \Sigma$

$$maxAlpha(P, \alpha) = \{\phi_{\Sigma'} \mid \phi_{\Sigma'} \in P \wedge \\ \neg\exists \phi_{\Sigma''} \in P : |\Sigma' - \alpha| < |\Sigma'' - \alpha|\}$$

This defines the subset of path properties that are the most different in terms of the number of unshared symbols with alphabet α.

Another notion of diversity can be defined in terms of the rejected symbol sequences for a given property by enumerating the set of strings that drive acyclic transition sequences leading from the property's start state to its trap state.

Definition 4.3 (Acyclic Trap Strings): Given ϕ_Σ,

$$traps(\phi_\Sigma) = \{\sigma \mid \sigma \in \Sigma^* \wedge \Delta(s_0, \sigma) = s_{trap} \wedge \\ \neg\exists i \neq j : \Delta(s_0, \sigma[i]) = \Delta(s_0, \sigma[j])$$

where $\sigma[i]$ denotes the prefix of σ of length i.

Trap strings can be used to order a set of properties in terms of number of new trap strings that a given property would add relative to a given set of properties.

Definition 4.4 (Max Trap Strings): Given $P, M \subseteq L$

$$maxTrap(P, M) = \{\phi \mid \phi \in P \wedge \\ \neg\exists \phi' \in P : |traps(\phi) - \bigcup \phi_m \in M : traps(\phi_m)| < \\ |traps(\phi') - \bigcup \phi_m \in M : traps(\phi_m)|\}$$

Given a cost estimate for monitoring a program relative to set of observed program events, $cost \colon \mathcal{P}(\Sigma) \rightarrow \mathcal{Z}$, calculated, for example, by profiling a small set of runs, one can select properties based on a given cost threshold.

Definition 4.5 (Cost Threshold): Given $P \subseteq L$, and a cost threshold value $T \in \mathcal{Z}$,

$$costThresh(P, T) = \{\phi_{\Sigma'} | \phi_{\Sigma'} \in L \wedge cost(\Sigma') < T\}$$

Finally, one can select the set of properties with the maximum cost.

Definition 4.6 (Max Cost): Given $P \subseteq L$,

$$maxCost(P) = \{\phi_{\Sigma'} | \phi_{\Sigma'} \in L \wedge \\ \neg\exists \phi_{\Sigma''} \in P : cost(\Sigma') < cost(\Sigma'')\}$$

B. Sampling Strategies

Figure 4 shows the common core of our cost-aware property sampling strategy. In general, given a threshold on monitoring cost (T), our sampling strategy operates as follows: while the threshold has not been reached, *select* the "best" remaining property from S, update the set of properties (M) and the remaining cost threshold ($cost$), filter the remaining candidate properties by removing the ones that are subsumed by monitored properties ($addVd(M, L)$) or are too costly, and continue.

Selection of properties can be performed in many ways; we explore several approaches based on the selection functions defined in Section IV-A. Randomly selecting a property from a given set, $select(S, \ldots) = rand(S)$,

```
PropertySample(L,T)
    cost = 0
    S = costThresh(L,T)
    while (cost < T ∧ S ≠ ∅)
        φ_Σ' = select(S,...)
        M = M ∪ {φ_Σ'}
        cost = cost + cost(Σ')
        S = costThresh(addVd(M,L), T − cost)
    return M
```

Fig. 4. General Property Sampling Strategy

yields the *basic* sampling strategy. Choosing properties with maximal alphabet diversity, $select(S, \ldots) = rand(maxAlpha(S, \bigcup \phi_\Sigma \in M : \Sigma))$, yields the *symbol* sampling strategy. Choosing properties with maximal trap string diversity, $select(S, \ldots) = rand(maxTrap(S, M))$, yields the *path* sampling strategy. Composition of selection functions can also be used. For example, $select(S, \ldots) = rand(maxAlpha(maxCost(S), \bigcup \phi_\Sigma \in M : \Sigma)))$, yields the *cost-symbol* sampling strategy.

In the next section we describe our current implementation toolset which constructs the entire lattice in order to support the empirical exploration of the richness of the full property space. Our property sampling strategy can, however, be implemented at a lower cost by encoding sets of properties without enumerating them. For example, $addVd(\{\phi_{\Sigma'}\}, L)$ can be encoded as any property whose alphabet has a non-empty intersection with $\Sigma - \Sigma'$. We are exploring symbolic encodings that permit efficient on-demand construction of property samples.

V. Evaluation

In this section, we assess the performance of our approach through the following research questions:

RQ1: How rich is the space of properties defined by the lattice? We would like to explore the degree to which the lattice enriches the space of original properties and how property violation detection varies with alphabet size.

RQ2: How effective are various sampling strategies in detecting violations while operating under overhead constraints? We would also like to learn how sampled lattice properties perform in comparison to the original properties.

A. Study Setup

To answer the research questions we required a representative artifact with a set of path properties that we could monitor, and clients that may use the artifact in ways that violate its defined properties.

Artifact. We selected *Hibernate*, an open source Java library that provides support for object persistence [17], as our artifact. Its core library, Hibernate Core, consists of 78 java KLoC, is downloaded thousands of times a day, and is used by hundreds of other open source projects (including the Apache web server).

We derived path properties for Hibernate following the guidelines described in [19]. We examined the Hibernate API and its documentation, identified recommended usage patterns (and anti-patterns) that could be translated into constraints, and then translated those constraints into properties represented by regular expressions. We focused on the usage of transactions and object processing API calls while considering the three possible states of objects in Hibernate (transient, persistent, and detached) [23], [17]. This resulting property set has 9 path properties that include 17 Hibernate's API calls appearing in the $Session$, $SessionFactory$, and $Transaction$ classes. Given $SessionFactory\ sf$, $Session\ s$, and $Transaction\ tx$, the path properties are:

1) `sf.openSession()` or `sf.getCurrentSession()` before anything
2) `sf.openSession()` before `s.close()`
3) `s.close()` or `s.disconnect()` after `sf.openSession()`
4) `s.close()` or `s.disconnect()` or `tx.commit()` after `sf.getCurrentSession()`
5) `s.beginTransaction()` before `tx.commit()`
6) `tx.commit()` after `s.beginTransaction`
7) `tx.beginTransaction()` or `tx.getTransaction` before any object related operations (e.g., `s.get(A)`)
8) `s.load(A)` or `s.get(A)` before any object changing operations (e.g., `s.update(A)` or `s.delete(A)`)
9) No object changing operations (e.g., `s.update(A)`) after `s.delete(A)` unless preceded by `s.get(A)` or `s.load(A)`

Lattice and Samples. To generate the enriched property space defined by the lattice, we built prototype tools that accept property specifications and generate L_ϕ. To use the tools, we first encoded the 9 original path properties as regular expressions (as in Figure 1). The tools then use the Laser FSA toolkit [20] to convert the regular expressions into FSAs, construct their product, and obtain $\top$. The resulting $\top$ has 81 states, where 21 of the states are accepting states. The tools enumerate all possible combinations of symbols (API calls), resulting in the generation of 131,071 sub-alphabets, and project $\top$ over each sub-alphabet to create an FSA per sub-alphabet. Finally, the trivial FSAs are discarded resulting in an L_ϕ with 106,340 non-trivial sub-alphabet properties. We also implemented a tool to apply the sampling strategies from Section IV on this lattice to produce a property sample that can be fed as input to our object-sensitive FSA run-time monitoring framework [4].

Clients. We selected three open source applications as clients that utilize Hibernate: 1) an online auction application (AS) that comes with the Hibernate package

Client	# of Test Cases	Coverage	API Calls	Overhead	
				$\top$	$Orig$
AS	57	67%	10115	19%	28%
WS	71	77%	11781	20%	33%
NC	73	71%	8925	15%	23%

TABLE I. Summary of the three clients.

to illustrate its usage [23], 2) a WebStore (WS) online shopping application [24], and 3) NoteCat (NC) a web based application to manage Wiki notes [25]. The average client size is 4.2 KLoC. To drive the execution of the clients, we use an enhanced version of their test suites. We added tests to the original suites to exercise the clients more extensively so that, in turn, they would execute their calls to the Hibernate API, allowing us to check for path property violations.

To simulate usage violations of the Hibernate path properties, we generated mutations of the clients applications exhaustively using all the mutation operators available in muJava [26] and Jester [27] tools. We constrained the mutation generation to the client classes that are more closely related to the Hibernate operations. We then ran the test suite on each mutated version of each client and collected a trace of the calls made to the Hibernate API. We retained the mutants that generated a unique trace that violated $\top$ (including mutants that do not lead to a property violation would obfuscate the differences among property sampling strategies). As a result of this process, we retained 17 mutants for AS, 14 for WS, and 10 for NC.

Monitoring. We instrumented Hibernate to capture the execution of the 17 API calls that are part of $\top$, and to perform on-the-fly property monitoring.

We computed the monitoring overhead by comparing the time to execute a client's test suite on the original Hibernate versus the instrumented Hibernate. The test suite attributes and the overhead for each application when running the complete test suite is presented in Table I. Column 3 shows the percentage of methods covered by the test suites, column 4 gives the number of API calls observed during the execution, and column 5 and 6 report the monitoring overhead corresponding to the number of observed calls for the $\top$ property and the 9 $orig$inal properties, respectively.

Variables. We manipulate three independent variables. In both questions, we analyze the effects of changing the **space of path properties to monitor.** We can monitor the set of original properties ($orig$) and the ones in the lattice (lat) constructed from $orig$. To answer the second research question, we manipulate the **overhead bound** by setting an upper limit for monitoring overhead of 20%, 15%, 10%, 5%, and 1%. We also manipulate the **sampling strategy**. We consider the four sampling strategies described in Section IV: $basic$, *cost-symbol*, $symbol$, and $path$, where the cost estimate of each property considered by the sampling strategies is obtained by profiling the run of each client application's test suite.

We account for two dependent variables: cost and effectiveness. We use the number of events as a proxy for monitoring cost, since it is not dependent on the monitoring implementation. We measure monitoring effectiveness in terms of the number of unique violations detected.

B. Results for RQ1

To explore the diversity of the space of properties defined by the lattice, we analyze how each mutant of each client utilizes Hibernate when their corresponding test suite is executed. The analysis consists of checking what API calls are made (to approximate cost) and what properties are violated (to measure effectiveness)

Figure 5 provides a scatter plot for each client characterizing the relationship between the size of the monitored properties and their violation detection power.

The first thing we notice is how the population of properties defined by the lattice covers a wide range in size and violation detection power, and how larger properties tend to identify more violations than smaller ones. For example, in AS, the best properties of size 3 can detect 10 of 17 violations, some properties of size 11 can detect all violations, and all properties of size 16 will detect at least 12 violations. Although there are differences in how the clients violate the API path properties (e.g., monitoring one of the properties of size 4 is enough to detect all violations caused by WS but a minimum property size of 10 is necessary to detect all the violations in AS), these tendencies hold for all clients.

Compared with the 9 properties in $orig$ (marked with $\times$ in the figure), the properties in lat always include a property of the same size that has greater (in 25 of 27 cases across the three clients) or equal (properties of size 2 and 3 in NC) violation detection power. This is due to the properties in lat having enforced additional constraints from projecting the common symbols from multiple $orig$ properties (such as in the case of properties with `s.close()` and `sf.openSession()` symbols) or from projecting different set of symbols comprising from multiple $orig$ properties. We also observe that, as properties grow in size, more properties in lat perform better than the $orig$, indicating that the opportunities for sampling are greater in the presence of more complex properties.

Similarly and as expected, we observed that as the size of properties increases, the monitoring costs tended to increase as well. For AS, monitoring properties for sub-alphabets of size 2 has an average cost of 1445 events, for size 9 the average cost is 5418, and monitoring the

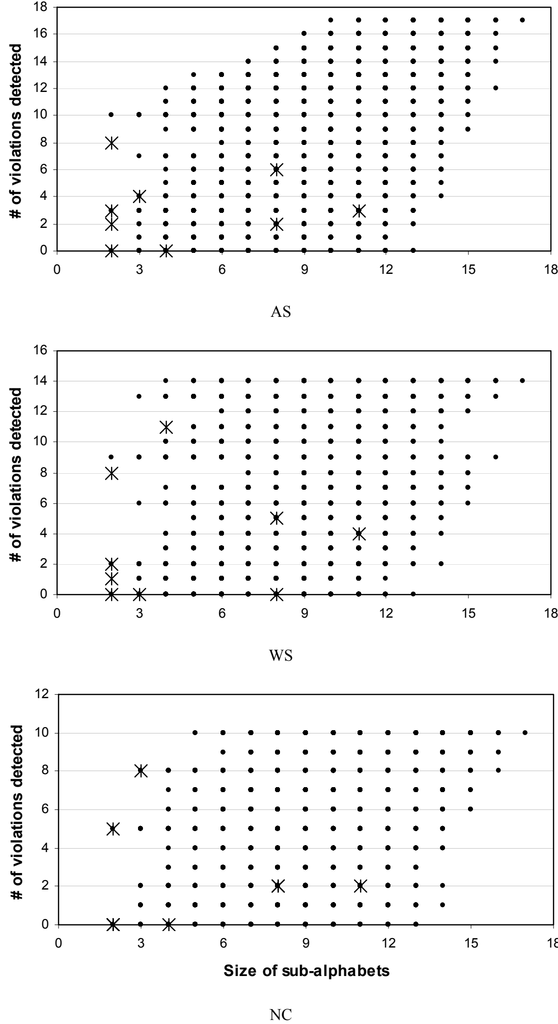

Fig. 5. The size of the sub-alphabets versus violation detection power for the necessary properties space of AS, WS, and NC. $*$ indicates a property in $orig$.

property of size 17 ($\top$) implies collecting 10115 events (due to space limitations the plots are not included). We also found that there is great variation in the monitoring cost across property sizes. For AS, monitoring a property including 3 API calls can result in observing between 969 and 2771 events. For WS and NC, this range can go up to 6000 events. As with the violation detection figures, lat offered many properties with a wide range of cost and violation detection power from which to sample, which we further analyze in the next section.

C. Results for RQ2

We now quantify the effectiveness of several path property sampling strategies when selecting a sub-set of properties to monitor from the 9 properties in $orig$ and the ones in lat, operating under bounded monitoring overheads. Since our property sampling strategies operate with a degree of randomness, we performed 10 runs and averaged the number of violations detected across them.

Figure 6 shows the box plots of the violation detection capability of the sampling strategies with AS under monitoring overhead constraints of 20%, 15%, 10%, 5%, and 1%. We only show the $basic$ and $symbol$ sampling strategies for the properties in $orig$, but we show all the sampling strategies for the lat property space.

Independent of the monitoring overhead bound, sampling over the space of lat consistently resulted in the detection of more violations than sampling over $orig$. This was more obvious with higher overhead bounds. For example, with an overhead bound of 20% and sampling with the $symbol$ strategy, the difference in performance between lat and $orig$ is up to 6 violations (35% increase). As the overhead bounds gets tighter, the differences among sampling strategies decreases because the number of properties fitting the overhead constraint is reduced. Furthermore, under extreme overhead constraints, the population of selectable properties is small enough that sampling strategies cannot make a difference.

When comparing the sampling strategies over the lat space of properties, we note that $path$ performs best. With an overhead bound of 20%, $path$ detects an average of 7 more violations than $basic$. As observed before, however, the benefits decrease under tighter monitoring overhead bounds (using $path$ leads to the detection of only 1 additional violation when the overhead bound is 5%.)

The box plots also reveal that the most advanced sampling strategies provide smaller violation detection variability (smaller boxes). This is beneficial because their performance is more consistent and can be better estimated than the $basic$ strategy. The reason for this low variability is that these sampling strategies aim to select the integrated property $\top$ (or the closest to it) which always provides the highest symbol and path coverage.

Due to space constraints and given the similar tendencies, for the WS and NC clients we only present the plots (Figure 7) for the $basic$ and the best performing sampling strategy over the space of properties defined by lat with 20%, 10% and 1% overhead bounds. For both clients, the most advanced sampling strategy detects more violations than $basic$. The gains, however, are more noticeable in WS than in NC. Note also how the best sampling strategy for WS is *cost-symbol*, while for NC is $symbol$. This difference may have been caused by how each client uses Hibernate and by their generated mutants. Most properties violated in WS include calls to commonly executed (and hence costly) symbols such as $beginTransaction$, $getTransaction$, and $commit$. On the other hand, the violations in NC involve a larger variety of symbols diminishing the impact of event monitoring cost.

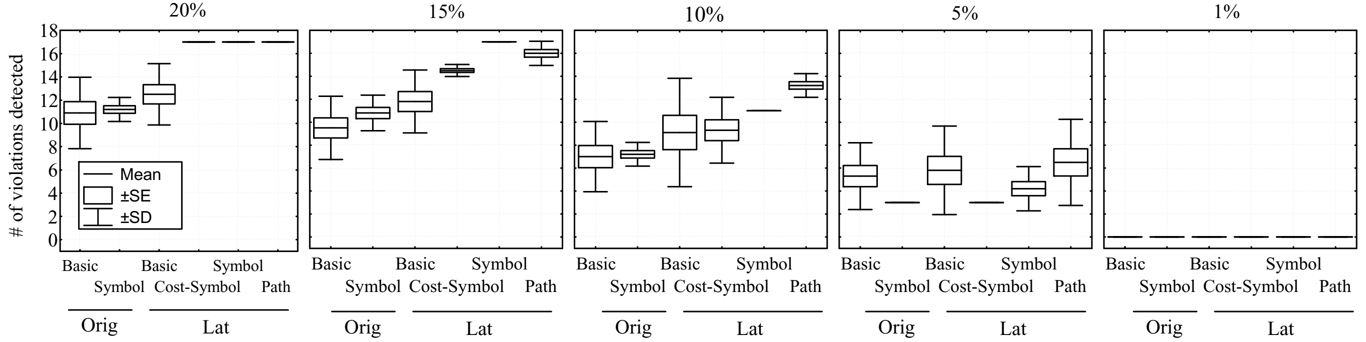

Fig. 6. Results for AS

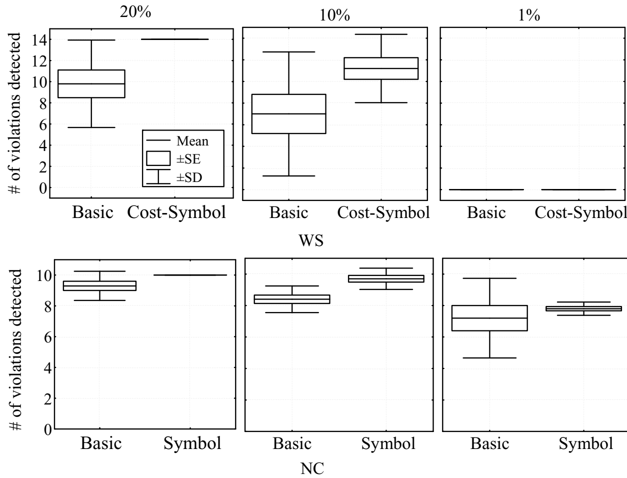

Fig. 7. Results for WS and NC

VI. Related Work

We summarize strategies aimed at increasing the efficiency of monitoring for *state properties*, and discuss how they have been adapted to work on *path properties*.

The first strategy consists of minimizing the number or improving the location of the probes necessary to profile the events of interest. These techniques attempt to avoid inserting probes that can render redundant or inferable information. This analysis can be performed off-line [6], [7] or online, where probes are removed during a program execution when certain conditions are met [8], [28], [29], [9], [30].

The removal of a probe trivially preserves the soundness of violation reports for state properties monitoring. However, inferring meaningful sequence of observations for path property monitoring may impose dependencies between multiple probes. Consequently, removing a probe may cause an analysis to become unsound. Recent work addresses this issue by applying sophisticated static analyses to the path property under analysis to adjust the monitoring probes offline [2], [15], by employing a watchdog thread to periodically trigger simultaneous enabling/disabling of probes [31] (this approach may miss a violation), or by calculating and enabling only the set of probes necessary to observe the transitions to other states [4]. These techniques are orthogonal to and can be applied jointly with our approach. We will explore their combination in future work.

Another widely used strategy involves sampling to select only a subset of probes to reduce monitoring overhead with an expected loss of accuracy. The effectiveness of sampling depends on the sample size and the selection strategy. Probes can be sampled across multiple dimensions, such as time [13], population of events [11], [14], or deployed user sites [12], [32], while their selection strategies range from basic random sampling (used by many of the commercial and open source tools), conditionally driving the observation paths based on a predefined distribution [14], or stratified proportional sampling on multiple populations [12]. The flexibility offered by the various sampling schemes makes them very amenable for profiling activities that can tolerate some degree of data loss.

The only sampling approach we are aware of that addresses the challenge of preserving soundness of violation reports in path property monitoring proposes the identification of regions of program behavior that correspond to independent *instances* of a path property; and then samples across those instances [31]. This approach combats, to a certain extent, the non-trivial component of monitoring cost that is due to the number of program objects that must be monitored [1]. Our approach also utilizes sampling, but is distinct from the previous approach in that it provides a richer population to sample, where each sub-alphabet property exposes a tradeoff between monitoring cost and violation-detection. Moreover, our sampling strategies consider the lattice structure of the sub-alphabet properties to obtain a diverse set of properties such that each sampled property can potentially detect a disjoint set of path violations from every other sampled properties. Bodden's approach could be combined with our lattice-based sampling approach to produce, in effect, a finer lattice that expands each property into a cluster of instances. Our approach is also the first one to be assessed in terms of violation detection effectiveness, which is

particularly important for sampling techniques since they may miss violations.

VII. Conclusions

We have introduced a novel approach for controlling the overhead cost of monitoring path properties by composing a single integrated property from a set of simpler and smaller properties, decomposing it into a set of sub-alphabet properties that collectively preserve the integrated property's violation detection power, and then sampling a subset of these sub-alphabet properties to enable monitoring under tight overhead constraints. We conducted a preliminary study, employing various sampling schemes that leverage the lattice of sub-alphabet properties. Our results show the potential of our approach to detect more path violations with less overhead when compared to monitoring the original properties.

Next, we are interested in adapting our approach to perform continuous path property monitoring, especially in the context of deployed programs. We believe that the population of sub-alphabet properties in the lattice offers an excellent opportunity to guide and manage the parallelization of monitoring activities across deployed sites. We also want to investigate iterative sampling schemes that take into account the feedback from previous observations to tailor the monitoring activities, for example, based on the real usage pattern and cost of the monitored sites. Finally, we want to perform further studies to evaluate the effectiveness of our approach over other complex path properties.

References

[1] P. Avgustinov, J. Tibble, and O. de Moor, "Making trace monitors feasible," *SIGPLAN Not.*, vol. 42, no. 10, pp. 589–608, 2007.

[2] E. Bodden, L. Hendren, and O. Lhotak, "A staged static program analysis to improve the performance of runtime monitoring," in *21st Euro. Conf. on Obj.-Oriented Prog.*, 2007, pp. 525–549.

[3] F. Chen and G. Roşu, "Mop: an efficient and generic runtime verification framework," in *Proc. 22nd Conf. on Obj. Oriented Prog. Sys. and App.*, 2007, pp. 569–588.

[4] M. Dwyer, A. Kinneer, and S. Elbaum, "Adaptive online program analysis," in *Int'l. Conf. on Softw. Eng.*, 2007, pp. 220–229.

[5] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. V. Sokolsky, "Java-MaC: A run-time assurance approach for Java programs," *Formal Meth. Sys. Design*, vol. 24, no. 2, pp. 129–155, 2004.

[6] H. Agrawal, "Efficient coverage testing using global dominator graphs," in *Works. on Prog. Anal. for Softw. Tools and Eng.*, 1999, pp. 11–20.

[7] T. Ball and J. R. Larus, "Efficient path profiling," in *Int'l. Symp. on Microarchitecture*, 1996, pp. 46–57.

[8] K.-R. Chilakamarri and S. Elbaum, "Reducing coverage collection overhead with disposable instrumentation," in *Int'l. Symp. Softw. Rel. Eng.*, 2004, pp. 233–244.

[9] J. Misurda, J. A. Clause, J. L. Reed, B. R. Childers, and M. L. Soffa, "Demand-driven structural testing with dynamic instrumentation," in *Int'l. Conf. Softw. Eng.*, 2005, pp. 156–165.

[10] C. C. Williams and J. K. Hollingsworth, "Interactive binary instrumentation," in *Int'l. Works. on Remote Anal. and Measurement of Softw. Sys.*, May 2004, pp. 312–327.

[11] M. Arnold and B. G. Ryder, "A framework for reducing the cost of instrumented code," in *Conf. on Prog. Lang. Design and Impl.*, 2001, pp. 168–179.

[12] S. G. Elbaum and M. Diep, "Profiling deployed software: Assessing strategies and testing opportunities." *IEEE Trans. Softw. Eng.*, vol. 31, no. 4, pp. 312–327, 2005.

[13] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," in *Symp. on Compiler Construction*, 1982, pp. 120–126.

[14] B. Liblit, A. Aiken, and A. Zheng, "Distributed program sampling," in *Conf. on Prog. Lang. Design and Impl.*, 2003, pp. 141–154.

[15] M. Dwyer and R. Purandare, "Residual dynamic typestate analysis exploiting static analysis: results to reformulate and reduce the cost of dynamic analysis," in *Int'l. Conf. on Aut. Softw. Eng.*, 2007, pp. 124–133.

[16] T. Ball, O. Kupferman, and G. Yorsh, "Abstraction for falsification," in *Proceedings of the 17th International Conference on Computer Aided Verification*, 2005, pp. 67–81.

[17] http://www.hibernate.org.

[18] http://java.sun.com/j2se/1.4.2/docs/api/java/nio/-channels/SocketChannel.html.

[19] M. Dwyer, G. Avrunin, and J. Corbett, "Patterns in property specifications for finite-state verification," in *Int'l. Conf. on Softw. Eng.*, May 1999, pp. 411–420.

[20] http://laser.cs.umass.edu/tools/.

[21] G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," in *Proc. 29th Symp. on Princ. of Prog. Lang.*, 2002, pp. 4–16.

[22] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[23] C. Bauer and G. Ling, *Java Persistence with Hibernate*. Greenwich, CT: Manning Publications Co., 2007.

[24] http://sourceforge.net/projects/webstore-app/.

[25] http://sourceforge.net/projects/notecat/.

[26] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "Mujava: an automated class mutation system: Research articles," *Softw. Test. Verif. Reliab.*, vol. 15, no. 2, pp. 97–133, 2005.

[27] http://jester.sourceforge.net/.

[28] http://profiler.netbeans.org/.

[29] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Conf. on Prog. Lang. Design and Impl.*, 2005, pp. 190–200.

[30] M. M. Tikir and J. K. Hollingsworth, "Efficient instrumentation for code coverage testing," in *Int'l. Symp. Softw. Test. Anal.*, 2002, pp. 86–96.

[31] E. Bodden, L. J. Hendren, P. Lam, O. Lhoták, and N. A. Naeem, "Collaborative runtime verification with tracematches," in *Workshop on Runtime Verif.*, 2007, pp. 22–37.

[32] A. Orso, D. Liang, M. J. Harrold, and R. Lipton, "Gamma system: continuous evolution of software after deployment," in *Int'l. Symp. Softw. Test. Anal.*, 2002, pp. 65–69.