Computer Languages, Systems & Structures 47 (2017) 170–188

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

Refinement of structural heuristics for model checking of concurrent programs through data mining

Reed Milewicz*, Peter Pirkelbauer

University of Alabama at Birmingham, Birmingham, AL 35223, United States

ARTICLE INFO

Article history:
Received 4 December 2015
Received in revised form
19 May 2016
Accepted 10 June 2016
Available online 18 June 2016

Keywords:
Model checking
Structural heuristics
Data mining

ABSTRACT

Detecting concurrency bugs in multi-threaded programs through model-checking is complicated by the combinatorial explosion in the number of ways that different threads can be interleaved to produce different combinations of behaviors. At the same time, concurrency bugs tend to be limited in their scope and scale due to the way in which concurrent programs are designed, and making visible the rules that govern the relationships between threads can help us to better identify which interleavings are worth investigating. In this work, patterns of read–write sequences are mined from a single execution of the target program to produce a quantitative, categorical model of thread behaviors. This model is exploited by a novel structural heuristic. Experiments with a proof-of-concept implementation, built using Java Pathfinder and WEKA, demonstrate that this heuristic locates bugs faster and more reliably than a conventional counterpart.

1. Introduction

For the purposes of this work, we define a *thread* as a sequence of instruction blocks that are linked to one another by a chain of continuations. A thread is the smallest unit of execution that can be handled independently by a scheduler, and threads are presented here as a general-purpose solution for managing concurrency. A multi-threaded program executing on a multi-core architecture consists of the birth, life, and dissolution of arbitrarily many threads operating in parallel across a fixed number of cores. In the analyses of multi-threaded, concurrent programs, our task is to characterize interactions between threads and other threads, and between threads and the execution environment. Meanwhile, *concurrency bug* is an umbrella term for classes of bugs which arise as a consequence of improper synchronization between threads over the use of shared resources, such as deadlocks, race conditions, and atomicity violations. Our goal is to detect concurrency bugs that could interfere with the proper execution of a concurrent program through careful analysis.

Recent history is replete with high profile news stories of concurrency bugs causing dramatic and spectacular failures. One especially powerful example, the Northeast blackout of 2003, a widespread blackout that left over 55 million people in the Northeastern United States and Canada without power for days, was triggered by a sequence of unsynchronized writes to a data structure in an alarm reporting system resulting in data corruption; locating the source of the bug required 8 weeks of analysis of over several million lines of code. Meanwhile, on the development side, there is an abundance of both anecdotal and empirical evidence that concurrency bugs are a fairly common and onerous occurrence. For example, a 2007 publication based on an internal survey of the use of concurrent programming at Microsoft found that concurrency bugs were a frequent

* Corresponding author.
E-mail addresses: rmmilewi@cis.uab.edu (R. Milewicz), pirkelbauer@uab.edu (P. Pirkelbauer).

http://dx.doi.org/10.1016/j.cl.2016.06.001
1477-8424/© 2016 Elsevier Ltd. All rights reserved.

issue for developers [25]. 66% of survey participants (armed with an average of 10.48 years of programming experience) reported having to deal with concurrency issues in their work. The majority of those participants reported spending time debugging concurrent code at least once a month or more, and the majority of concurrency bugs were ranked as being very severe, hard or very hard to reproduce, and multiple days (less than a week) were needed to find and correct them.

Many different tools and techniques exist to detect concurrency bugs. These include both automated techniques such as static analysis, coverage testing, and execution capture/playback tools, as well as development strategies such as pair programming, code inspection, and the use of rigorous code standards. In addition to these, model-checking has proven to be a powerful and versatile approach for the task. The model checking problem can be summarized as such: given a state-transition graph of a system and a specification, prove that the system conforms to or "models" that specification. A model-checking engine automatically and exhaustively explores the state space of the system, and either verifies that all states satisfy the constraints of the specification, or finds a counterexample by showing that it is possible to arrive at an invalid state, one that violates the specification. The strengths of model-checking are that it can simultaneously verify arbitrarily many correctness properties in the course of its exploration, and that it can precisely identify the when, where, and how violations occur. These qualities are especially important for the detection of concurrency bugs, which can come in many forms and may only manifest themselves under very specific conditions. However, its strengths play into its key weakness: the state spaces of multi-threaded, concurrent programs can be inconceivably vast due to the sheer number of different ways that individual threads can be interleaved with one another. Finding ways to reduce and manage this combinatorial explosion remains an active area of research; as an influential example, we direct readers to Flanagan et al. [22] which introduced the concept of dynamic partial-order reduction (DPOR), and to Sistla et al. [47] which gives a comprehensive survey of symmetry reduction techniques prior to the development of DPOR. In traditional model-checking, state space exploration is blind and undirected. This is because the original focus of model-checking research was on total verification, and therefore all states must be explored. In contrast, directed model checking, which this paper focuses on, explores states in an order determined by a heuristic function, the assumption being that invalid states exist and an appropriate heuristic will help us flush them out quickly so as to provide counterexamples. As noted by Bhadra et al. [3], directed model checking is a form of *hybrid verification* that uses state space search as a platform for combining formal analysis with "informal" heuristics.

Those who investigate state space search heuristics, those who spend a great deal of time and energy developing and fine-tuning them, savor a delicious irony: all reasonable heuristics perform just as well as any other *on average*. That is to say that for any problem instance for which a particular heuristic provides optimal performance, we can contrive arbitrarily many other problem instances for which another heuristic is better suited; a formal treatment of this issue can be found in the well-known work by Wolpert and Macready [51] on the No Free Lunch Theorem (NFLT). The goal then is to identify which heuristic (or combination of heuristics) will provide the best performance for a given problem instance or class of problem instances. This has motivated deep investigations into general-purpose and highly parameterizable heuristics, meta-heuristics, and meta–meta-heuristics which allow users to tailor existing approaches to novel problems. However, as is illustrated in Fig. 1, every additional layer of optimization provides diminishing returns: each consecutive layer optimizes the parameters of the last to refine the information we receive, adding a modicum of overhead and creating an even more sensitive parameterization problem to solve. Furthermore, because our heuristics, the underlying source of our observations, are subject to the gravitational pull of the NFLT, we cannot reliably achieve the escape velocity needed to reach near-optimal resource consumption.

Then again, that zero-sum limit only applies if the distribution of programs with respect to errors is uniform; that is, it is impossible to make assumptions that hold true for most programs. Empirical studies of concurrent programs show that this position is overly conservative. Case in point, a 2008 comprehensive study of real-world concurrency bugs by Lu et al. [35]

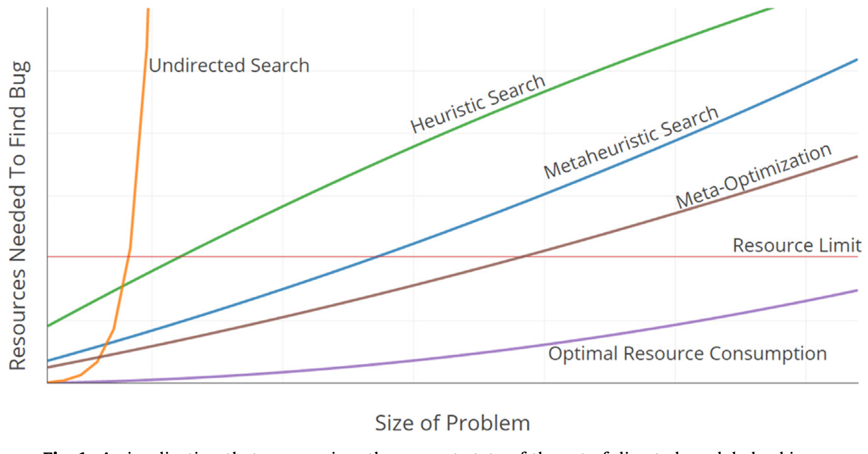

Fig. 1. A visualization that summarizes the current state of the art of directed model checking.

suggests that while concurrency bugs are becoming more common, they are not increasing in severity. Among their observations of concurrency bugs that occur in popular, large-scale programs, we would like to note the following:

- 96% of concurrency bugs involve only *two* threads.
- 66% of concurrency bugs involve concurrent accesses to only *one* variable.
- 92% of concurrency bugs involve no more than *four* memory accesses.
- 97% of non-deadlock concurrency bugs are either atomicity violations or order violations.
- 97% of deadlock concurrency bugs involve *two* threads circularly waiting for at most *two* resources.

In short, concurrent programs are getting bigger, but the scope and scale of bugs are staying the same. This may seem mysterious at first, but it makes sense when we consider how threads are typically employed in programs. When threads share resources, it is because they have overlapping responsibilities and interests; they often execute similar code and have similar local states. It is possible for a program to put 100 threads to work on 100 completely different tasks, but more often than not, those 100 threads are performing only a handful of distinct tasks. Applying the pigeonhole principle, we know that past a certain point adding additional threads results in more of the same behaviors rather than novel ones. As such, the overwhelming majority of concurrency bugs can be observed by examining a relatively small number of threads and memory accesses. This observation is echoed by Mercer and Rungta [43], who demonstrated that the number, types, and categories of threads that can cause an error are key determinants of the error distribution and density of a state space and thus the difficulty of finding a bug.

If we have access to “special knowledge” about the program in question (or about concurrent programs in general), we can short-cut regions of the state space that are unlikely to contain a concurrency violation. This suggests that combining model checking with prior analysis of source code, byte code, execution traces, comments, documentation, and formal specifications may yield insights that could greatly improve our performance. However, there are several challenges that must be confronted with combining model checking with any other kind of analysis.

First, while prior analysis can elucidate the nature of potential bugs, translating that knowledge into a form that a heuristic can use is non-trivial. For example, static analysis could suggest that two lines of code may cause a race condition when interleaved, but that alone does not tell us how to reach a state where those lines of code are concurrently executed; incorporating additional analyses into the model checking process introduces a new decision space with complicated trade-offs. Second, the computational cost of the prior analysis must either be low enough to be offset by savings from the reduction in the cost of the state space search, or the information must be reusable so as to amortize the costs. It is because of these challenges that the hybridization of model checking with other analyses remains an under-explored area of research.

In this work, we explore both a general claim about solutions to these challenges as well as a specific claim which we test by offering a proof-of-concept implementation. Our general claim is that by using techniques from data mining and machine learning, we can construct cost-efficient “soft” analyses that extract from a modality of a program structural information that can be mapped to the state space. Our specific claim is that framing the behaviors of threads and their relationships in a categorical way, that is, by reasoning about their behaviors in aggregate, could help us to predict what combinations of behaviors are likely to lead to invalid or erroneous states, allowing us to seek out those states more efficiently.

The paper presents the following contributions: (1) a lightweight technique for modeling functional similarities between threads based on stochastic approximations of read/write behaviors; (2) a novel structural heuristic which leverages these models to explore the state space of a concurrent program more efficiently; (3) a mutation-based benchmark generation process for the fair evaluation of heuristics. The paper is outlined as follows: Section 2 presents background information and related work. Section 3 describes our implementation in detail, and Section 4 discusses how we tested our heuristics and the obtained results. Section 5 discusses related work. Finally, Section 6 summarizes the paper and discusses possible future research directions.

2. Background

2.1. Structural Heuristics

Much work has been done on developing classes of heuristics that target particular kinds of bugs and violations. For a given state, the heuristic measures the likelihood that that state will either contain a violation or lead us to a violation. Many heuristics are designed to target specific kinds of concurrency bugs, such as deadlock detection heuristics. However, works by researchers Groce and Visser [26,27] introduced the notion of a structural heuristic, which, instead of explicitly focusing on particular bugs, aim to expose a variety of bugs by favoring states that are most revealing of the behavior of the program, allowing the search algorithm to uncover bugs before exhausting the limits of memory. Structural heuristics are not intended to remove the need for heuristics that are tailor-made for a particular program, but they do provide an alternative when the nature of the bugs is unknown or several different classes of bugs may be present in the program.

For race condition detection, one especially effective heuristic developed by the authors is to favor paths through the state space that maximize the interleaving of threads. This is to say that the search algorithm is directed to explore regions of the state space in which threads are switched out as often as possible, which the authors claim exposes “the dependency of the threads on precise ordering” [26]. To implement this heuristic, we keep a limited history of the threads that have been scheduled to run. When a new thread is scheduled, we append that information to the history, and make a pass over the history; the heuristic value is computed by counting the number of times that the current thread has been scheduled and the number of other live threads that could be scheduled instead – the higher the value, the less "interesting" the path is to the search algorithm. If the search algorithm is given a choice between two or more states to explore next, it will explore the state where the heuristic value is lowest. We will refer to this heuristic as h_{TI}, and the following is a formal definition for reference:

$$h_{TI}(path, limit) = \sum_{i\,=\,path.length-limit}^{path.length} \begin{cases} path.length * N_{aliveThreads} & \text{if } path.get(i).tid = path.get(path.length-1).tid \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

Because h_{TI} maximizes the number of different interleavings of threads that we explore, it excels at uncovering bugs resulting from race conditions; intuitively, having as many live threads interacting in as many different ways as possible increases the chance that one thread may harmfully interfere with the operations of another. However, this heuristic can struggle at high thread counts because we are overwhelmed by an exponentially large number of equally "interesting" interleavings and only a small fraction of these will actually lead us to a bug. This indicates that we may be able to improve the performance of this heuristic by providing additional information that it can use to help break ties. We would like to take this opportunity to stress two points. First, that this problem is not unique to h_{TI}: it is common for heuristics that are general-purpose do so at the expense of discriminatory power. Second, that h_{TI} performs just as well as any other heuristic on average, which means that it is as just good of a place to begin our exploration as any other; there is an analogous discussion to be had for any other such heuristic.

It is here that we consider a thought experiment to evaluate the effectiveness of h_{TI} in light of the empirical evidence we presented earlier regarding the nature of concurrency bugs. Assume that we have an implementation of a producer–consumer queue that exhibits a concurrency bug, with $N_{producers}$ threads producing values and $N_{consumers}$ threads consuming them $((\|threads\|) = N_{consumers} + N_{producers})$. Using $A*$ search guided by h_{TI} with a history limit of L, we conduct a search of the state space. Assuming $L < \|threads\|$, there are up to $\|threads\|^L$ many possible histories of length L, and of these, there exist $\|threads\| * (\|threads\| - 1)^{L-1}$ histories where the most recent thread has been scheduled only once (i.e. where $h_{TI}(path, L) = 0$). For a fixed L, we know that

$$\lim_{\|threads\| \to \infty} \frac{\|threads\| * (\|threads\| - 1)^{L-1}}{\|threads\|^L} = 1 \tag{2}$$

which explains why we often observe a proliferation of the number of optimally interesting states as the count of threads increases. Intuitively, we can compensate by increasing the size of L, thereby increasing the amount of information available to the heuristic to better distinguish between alternatives. However, increasing L has the effect of deepening and narrowing the search, which can cause us to miss buggy states at shallower depths; in our experience, even modest increases in L (e.g. 8–64) had the effect of slowing performance by a factor of up to 100.

Next, we know that 96% concurrency bugs involve only two threads, and of these bugs, there are three possible flavors: producer–producer, producer–consumer, and consumer–consumer. If we assume that these categories of bugs are equally likely to occur, then we arrive at two mutually exclusive hypotheses. The first hypothesis is that the bug occurs between threads executing different codes (producer–consumer bugs). If so, the best policy is to favor paths that treat consumers and producers equitably. This means exploring states whose recent histories are neither producer-dominated nor consumer-dominated, maximizing our chances of observing a producer–consumer violation. The second hypothesis is that the bug occurs between threads executing the same code (producer–producer or consumer–consumer bugs). If that were true, then we would want to favor paths that tightly interleaved producers with other producers and consumers with other

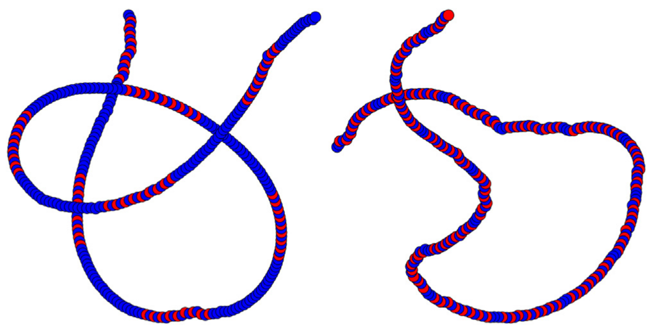

Fig. 2. A visualization of two partial paths through the state space of a program simulating operations on a producer–consumer queue. States where a producer is running are colored blue and states where a consumer is running are colored red. The path on the left schedules producers closely together with other producers and consumers with other consumers, and the path on the right aggressively interleaves producers and consumers. The heuristic values of these paths are *everywhere equal* according to the h_{TI} heuristic, but by our hypotheses, one of these paths is more likely to expose a violation than another. (For interpretation of the references to color in this figure caption, the reader is referred to the web version of this paper.)

consumers. As noted in Fig. 2, the h_{TI} heuristic cannot make these distinctions. This means that the heuristic can lead us to expend time and space following paths that are less likely to lead to a bug.

However, if we intend to make this information available for use, we must first impose an automatic mapping of threads to categories (e.g. producer, consumer) that we can exploit, and this takes the form of a clustering task.

2.2. Cluster analysis

Cluster analysis is the task of dividing a set of entities into disjoint subsets (clusters) such that entities within the same cluster are similar, and entities in different clusters are dissimilar (according to some measure). It is difficult to be more precise than this because the term applies to many different algorithms used in many different disciplines.

We begin by considering a clustering scheme as a manual process carried out by the programmer. If we take similarity to mean correspondence with the abstract design of the program, then the programmer, armed with full knowledge of the semantics of the program, labels each thread in the program according to their intended role or function. This partitioning can be used by heuristics to enrich their observations as we described in the previous subsection.

Entrusting this task to the programmer is an intuitive choice. The design of the program, made visible to us through the source code, is undergirded by what Dètienne [14] refers to as a “typology of schemas”, a mental framework for understanding the program that is difficult to automate. Despite this, we argue that automated inference of these categorizations is necessary. First, bugs are almost always the result of mismatches between the intended and actual semantics of the program, which means that the programmer may not have sufficient knowledge to accurately characterize the actual behavior of threads; here we refer the reader to Welles and Kodar [50] who argue that the original intent of the designer is both inaccessible and, in a practical sense, irrelevant. Second, the behavior of threads may be highly parameterized by input values, requiring careful analysis that can be burdensome on the programmer; we call to mind Engler and Musuvathi [19], a report on the use of model-checkers that found that for the sake of scalability and ease of use, a model-checker should “require as little input, annotations, and guidance as possible”.

Fortunately, there has been ongoing research in the fields of data mining and machine learning with regard to automated clustering techniques (see [32] for a comprehensive introduction to the topic). Clustering techniques have been extensively applied to problems in software analysis using features extracted from source codes, binaries, execution traces, and formal specifications. Popular applications include program comprehension [37,33,18] and anomaly detection and analysis [23,31,44,15].

3. Implementation

The central hypothesis of this work is that augmenting structural heuristics with knowledge of the semantics of threads and their interactions can help us identify concurrency bugs more quickly and at reduced cost. It is imperative that we qualify this hypothesis, because the phrase “knowledge of the semantics” is nebulous and belies the complexity of the task. We use the term semantics broadly to refer to the code that threads execute, the values that they read and write, and the way in which threads interact with one another insofar as that affects the first two items. By knowledge, we mean structured data, correlated with the semantics, that can be put to use by the heuristic. There are many design decisions that must be made as to how the semantics should be quantified, how the knowledge should be structured, and how the heuristics should exploit that knowledge, and it is beyond the scope of this paper to explore all of the possible combinations. What we present here is a proof-of-concept implementation that we have realized using Java Pathfinder [49], a stateful model-checker, and WEKA [28], a machine learning library developed at the University of Waikato.

3.1. Data collection

In this work, our approach uses read–write patterns of threads, extracted from execution traces of the program, to discriminate between classes of threads. Read–write sequences provide an abundance of context-free observations on which to build models; reads and writes are the most common and essential of operations, and their order and frequency reflect the code that a thread executes as well as hinting at the influence of other threads. We can represent the read–write behaviors of a thread over one or more executions by using a Markov chain model. A Markov chain is a discrete-time stochastic process that describes how a random variable changes over time, its possible values belonging to a countable set known as the state space of the chain. A Markov chain obeys the Markov property, that is, that the probability distribution of the state of the variable at time $t+1$ depends only on the state of variable at time t; it is for this reason that Markov processes are called “memoryless” processes. For a thread T_i, we define $X_{i,t}$ to be a random variable that represents the operation performed by T_i at time t. We use the term “time” here loosely: there can be many intervening operations aside from loads and stores which we choose not to observe, and different threads are in no way ordered with respect to one another by time. The range of values that $X_{i,t}$ can take is the state space $S=\{read, write\}$, and the probability distribution can be described by the following transition matrix M_i:

$$\begin{bmatrix} P_{read,read} & P_{read,write} \\ P_{write,read} & P_{write,write} \end{bmatrix} \tag{3}$$

where $P_{read,read}$ is the probability that T_i will read again after reading and so on. The initial probability distribution (describing $X_{i,t}$) is $Q_i = [q_{read}\ q_{write}]$, which dictate the probability that T_i will read first or write first. Given an execution trace, we can compute M_i and Q_i for the thread T_i in the following way:

$$P_{read,read} = \frac{N_{read,read}}{N_{read,write} + N_{read,read}} \quad q_{read} = \frac{N_{read}}{N_{read} + N_{write}} \tag{4}$$

where $N_{read,read}$ and $N_{read,write}$ are respectively the number of times that a read followed another read and the number of times that a read followed a write, and N_{read} and N_{write} are respectively the number of times that a read occurred in the trace and the number of times that a write occurred in the trace. The accuracy of this model can be improved by incorporating additional execution traces, which can be accomplished by computing the transition matrix and initial distribution for a thread in each trace and then averaging the results. In practice, we find that we reach the point of diminishing returns after just one trace, and therefore a single execution provides ample data on which to construct these models. The reason is that our Markov chains offer a very low resolution representation of thread behaviors, and therefore less data is needed. This resolution can be increased by increasing the order of the chain model, that is, tracking operations at the level of subsequences rather than individual operations (e.g. $P_{(read,read),(read,write)}$). However, doing so invites the noise that we mentioned earlier, and we would require much more data in order to construct accurate models.

3.2. Clustering

Once we have our thread behavior data, a collection of Markov chains, we can then begin the process of categorizing threads according to their behaviors. For this, we use k-means clustering. The k-means clustering technique attempts to partition a set of tuples $o_0 \dots o_n$, called observations or instances, into $0 < k \le n$ clusters so as to minimize the within-cluster sum of squares (WCSS), which is to say that it minimizes the squared Euclidean distance between each instance within each cluster. This k must be chosen by the user, and we discuss the consequences of this in the results section. For a thread T_i, we map the chain (M_i, Q_i) to a 4-dimensional vector $o_i = (P_{read,read}, P_{read,write}, P_{write,read}, P_{write,write})$ (note that we opt to discard the initial distribution). To perform k-means clustering, we use Lloyd's algorithm, which iteratively refines cluster assignments until convergence is reached [34]. For the sake of brevity, we will not elaborate on the details of the algorithm here. The end result of this process is that we have a mapping from thread identifiers to cluster assignments of the form $\{T_i = v_j : T_i \in identifiers, 0 < v_j \le k\}$.

3.3. Developing heuristics

Finally, once we have the partitioning of threads, our model of the underlying structure of the program, we can consider the implementation of heuristics that exploit that structure. As a guiding image, we can envision the k-means cluster assignments as an improper graph coloring of the state space, where each state is colored according to the cluster to which the currently executing thread belongs. The state space now has not only shape and form but also complex visual textures, providing additional criteria that we can use to compare partial paths. Intuitively, differences in cluster assignments (colorings) reflect differences in code and values. Exploring cluster-diverse (highly chromatic) paths favors behaviors that occur between different classes of threads, and exploring non-cluster-diverse (weakly chromatic) paths favors behaviors that occur between threads of the same class. With this in mind, we arrive at our first heuristic, h_{CI}, which we formally define as follows:

$$h_{CI}(path, limit) = \sum_{i = path.length - limit}^{path.length} \begin{cases} path.length * N_{aliveClusters} & \text{if } getCluster(path.get(i).tid) = getCluster(path.get(path.length-1).tid) \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

Threads that could be scheduled next are weighted according to the number of times that that thread's cluster has been scheduled in the recent history and the number of clusters that could be scheduled. This heuristic tends to perform poorly on its own, because it only encourages interleaving threads to the extent that all live clusters are represented in the recent history (i.e. at least k threads); any two threads from the same cluster are given the same weight, regardless of how often they have been scheduled in the past. This motivates us to incorporate both h_{TI} and h_{CI} into a single heuristic, which we will refer to as h_{TICI}:

$$h_{TICI}(path, limit) = h_{TI}(path, limit) \pm h_{CI}(path, limit) \tag{6}$$

By plus-or-minus h_{CI}, we mean "choose one". Our argument is that the value of h_{CI} is correlated with the distance to a buggy state involving a race condition (if one exists), but the mutually exclusive hypotheses we made in Section 2 imply that we do not know the *sign* of the correlation. That is, we claim that at least one version of the heuristic (which we will refer to as h^{+}_{TICI} and h^{-}_{TICI}) will almost always find bugs faster than h_{TI} alone, although it is possible that one version may do worse. This is intentional. The information provided by the cluster analysis is purely structural, but the heuristic is phrased such that we can incorporate information from additional analyses, which we will discuss in greater detail in Section 4. A diagram that summarizes the process that has been described in this section can be seen in Fig. 3.

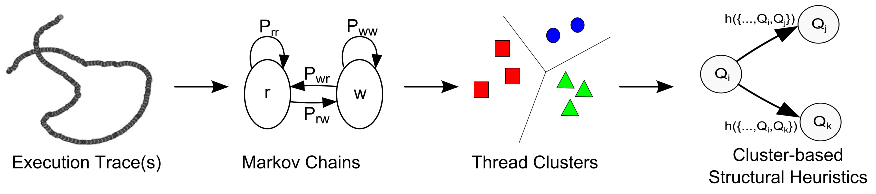

Fig. 3. An overview of the process described in Section 3. Executed traces are mined for read–write sequences belonging to each thread, with the results being used to train Markov chain models. K-means clustering is performed on the transition matrices of these models, providing a partitioning that informs the heuristic search.

4. Evaluation

To determine whether or not our behavior model and our heuristic are worthwhile, there are two criteria which they must satisfy. The first is that the automatically inferred behavior model agrees with and produces results of comparable quality to a manually constructed behavior model. The second is that our proposed heuristic must perform better than the state-of-the-art approach which does not take advantage of our behavior model. If our approach agrees with the manual approach but the performance results are poor, then either our model does not provide any meaningful benefit or our heuristics are inadequate, and if we have poor agreement but good performance, then our model works but our understanding of it is incorrect. We want to meet both of these requirements, and the experiments in this section are designed to test this.

4.1. Agreement and quality of cluster analysis

According to our high-level, abstract knowledge, each benchmark consists of N threads, one master thread and $N-1$ workers, and M classes of tasks (usually two) distributed evenly over the worker threads. We can partition the threads into categories that correspond with their role in the program, a clustering according to the designer's intent. For a given benchmark we can compute two clusterings, one using our manual method and the other using k-means clustering on Markov chain models. Due to the nature of the data on which we perform clustering, it is possible to produce misclassifications which can cause our automated approach to deviate from the manual one. For example, given a program consisting of multiple threads repeatedly attempting to perform a single compare-and-swap operation on the same memory location, the lone thread who takes the longest to succeed may be placed into a separate cluster. In practice, this is unlikely to happen for larger programs, and in our experiments, we found that occasional misclassifications did not significantly alter performance.

Our first task is to test whether or not the automated approach agrees with the manual approach, that is, that the two, despite being derived from different data, produce significantly similar results. For this, we compute their Rand index [41], a commonly used measure of cluster similarity. The Rand index compares the number of cluster assignments on which the two approaches agree to the number on which they disagree, and produces a real value from 0 to 1, with 0 indicating no agreement and 1 meaning the two agree perfectly.

Our second task is to compare the quality of the clusterings accordingly based on the data that was clustered – in machine learning and data mining parlance this is known as internal evaluation. This is complicated by the fact that we do not have access to the data on which the manual approach was performed, the source code being separate from the psychological schemas used to interpret that source code. However, we do have access to the Markov chain representation, and if our understanding is correct, then agreement of the two approaches indicates that the two underlying sources of data strongly correlate with one another. To measure the quality of the clusterings, we compute their homogeneity (H) and separation (S). Homogeneity is the average distance between each instance and the center (average) of the cluster that it belongs to, and separation is the weighted average distance between cluster centers. Ideally, H should be as small as possible, and S should be as large as possible.

To gather this data, experiments were performed on a selection of Java benchmarks belonging to the Software-Artifact Infrastructure Repository (SIR) of the University of Nebraska, which provides a wide variety of software objects written in Java, C, C# and C++ [16]. Of the Java benchmarks, we selected a subset of 37 which contains documented concurrency bugs, and within that category, we present results on 7 of those benchmarks. All of the benchmarks exhibit a division of labor between threads, arrangements which are amenable to our clustering strategy. The following is a summary of those benchmarks:

- **boundedbuffer**: An implementation of a bounded buffer that contains a deadlock based on a common bug pattern reported by Etyani and Ur [20].
- **clean**: Two groups of threads perform tasks that require each to wait on the signal of the other in order to proceed, which can result in a deadlock, based on a bug pattern reported in Farchi et al. [21].
- **losenotify**: One group of threads depends upon a notification from another in order to progress in such a way that invites deadlock. This benchmark, like **clean**, is based on work by Farchi et al. [21].

- **nestedmonitor**: A non-deterministic implementation of a bounded buffer that uses semaphores to negotiate access to the data structure. Deadlock can occur due to nested monitor calls. This particular benchmark was based on an example provided in Kramer and Magee [36].
- **readerswriters**: This benchmark showcases a race condition in which threads may begin writing while other threads are reading in an unsynchronized way, producing a race condition. This benchmark is based on the work of Pasareanu et al. [38].
- **reorder**: This benchmark demonstrates an atomicity violation, which is reported as a runtime exception, occurring between threads which read from two memory locations and threads which write to those memory locations. The benchmark is based on work by Farchi et al. [21].
- **twostage**: This benchmark involves a two stage access bug in which one group of threads attempts to write to two different memory locations, separately acquiring and releasing a lock on each, while another group of threads attempts to read from them producing a race condition. This kind of race condition is often observed with database transactions when a client fails to lock multiple relevant datasets simultaneously. The benchmark is based on work by Farchi et al. [21].
- **wronglock**: Two different groups of threads attempt to acquire locks to access data, but one group of threads consistently acquires the wrong locks, which results in a runtime exception. The benchmark is also based on work by Farchi et al. [21].
- **lfqueue**: A custom implementation of an unbounded lock-free queue. In the dequeue method, an atomic compare-and-set operation on the head of the queue has been replaced with an atomic load and atomic store, which can cause a thread performing a dequeue to enter into a race condition with any other thread also reading or writing to the location of the head.
- **prioritytree**: An implementation of a tree-based priority queue based on an implementation given in *The Art of Multiprocessor Programming* by Herlihy and Shavit [29]. In the original version, each node maintains a bounded, atomic counter that indicates the priority of the value it holds. In our version, the counter is made non-atomic, which invites race conditions.

4.2. *Benchmark generation and performance tests*

Our ultimate goal is to provide heuristics which find bugs in less time and with smaller memory requirements so that programmers can analyze their concurrent code more easily and frequently. Given a set of representative benchmarks, we can compare the performance of each heuristic by measuring the execution time and space needed to reach at least one state containing a concurrency error, while varying the number of threads (exponentially increasing the size of the state space). This is a form of weak scaling, which demonstrates how well the resource requirements of each search algorithm scale with the size of the input.

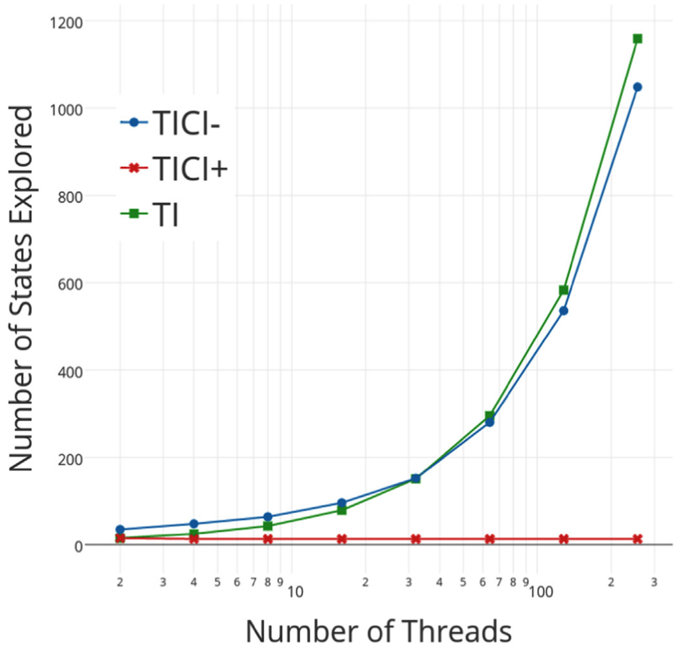

Fig. 4. Performance results at different thread counts for the *reorder* benchmark. These results reflect a general trend, in that one parameterization of h_{TICI} performs much better than h_{TI} and that another provides comparable (though possibly inferior) performance. But because benchmarks like these are small concurrency kernels, we tend to get results that are much better than what we should expect for real world applications.

Unfortunately, when attempting to test our approach empirically, we encounter a paradox: concurrency violations are common, yet benchmarks of concurrency violations are scarce by comparison; this is largely due to the labor costs involved in collecting and curating a benchmark suite. Some benchmark suites like that of Etyani and Ur [20] are constructed by manually engineering concurrency violations into otherwise healthy code. Others like that of Pasareanu et al. [38] rely on examples of real-world code that contain documented concurrency bugs. Finally, larger curated bug corpuses like the Software-Artifact Infrastructure Repository (SIR) of the University of Nebraska [16] assemble mixed collections of artificially constructed and real-world codes. Benchmark suites tend to be small (often less than 50 cases), but offer a wide range of different kinds of software and different kinds of concurrency violations. Focusing on breadth rather than depth is preferred when validating an analysis with respect to false/true positives and negatives. However, when comparing different analyses in terms of resource efficiency, a shallow suite of benchmarks can be misleading. In our preliminary experiments on the benchmarks previously mentioned, we found that our approach found bugs faster than the competition, but because our heuristic was designed with a specific class of concurrent programs in mind, the number of benchmarks we could test on was small, and this made the results questionable. An example of this kind of performance result can be seen in Fig. 4. We want to avoid any unintentional cherry-picking of benchmarks.

With that in mind, we turn our attention towards mutation analysis as a solution to our benchmark problem. In the context of software testing, mutation analysis is a method for using fault injection to measure the quality of a test suite. An artificial defect (a mutant) is introduced into a program to be tested, and if the test suite is capable of detecting the defect, the mutant is said to be *killed*, and is said to have *survived* otherwise. Mutations are generated according to a fault model, a “toolkit” of program modifications that have the potential to either create havoc or to cause the behavior of a program to deviate in subtle ways from its intended behavior. Studies by Andrew et al. [2] and Do and Rothermel [17] support the claim that mutation-induced faults can produce realistic facsimiles of hand-crafted and real-world buggy codes. Mutation analysis is a well-known technique within the software testing community in general and the Java development community in particular; a comprehensive 2011 survey by Jia and Harman [30] notes over a hundred published papers concerning mutation analysis of Java, and a 2013 survey by Delahaye and Bousquet [12] mentions that there are over ten different mutation analysis tools for the Java language alone. Furthermore, Delmaro et al. [13], Bradbury et al. [6] and Wu and Kaiser [52] cover mutation analysis for concurrency violations in particular. We merely need to repurpose this analysis for our use to generate benchmarks on which to test our approach.

Our mutation analysis framework is based the ConMAn suite of mutation analysis operators for concurrent Java as seen in Bradbury et al. [6], which was implemented using TXL, a programming language designed to support source code analysis and source-to-source transformation tools [8–11]. We faithfully reimplemented a relevant subset of this operator suite using the ROSE source-to-source translation infrastructure developed at the Lawrence Livermore National Laboratory (LLNL) [40,45]. ROSE parses the input program and exposes to our tool a rich abstract syntax tree (AST) to which we can make modifications, and the resulting AST is then unparsed by ROSE and a new, mutated program is produced. A list of the mutation operators supported by our framework can be seen in Table 1. An auxiliary objective of this work is to explore the effectiveness of mutation analysis for benchmark generation, and we comment on this in the results.

In order to increase the destructive potential of our framework, we introduced several new mutation operators, including weakening strong atomic operations, and demoting atomic types to non-atomic equivalents (e.g. `AtomicInteger` and `Integer`). We also took advantage of ROSE's robust support for the Java language to extend the functionality of certain operators. In particular, the `EAN` mutation operator, which previously only handled `getAndSet` method for classes in the

Table 1
Mutation operators supported by our mutation analysis framework.

Mutation operator	Description
MSP	Modifies/replaces the parameter of a synchronized block
ESP	Exchanges the parameters of two synchronized blocks
EAN[a]	Divides an atomic call (e.g. CAS) into a non-atomic set of calls
WAO[b]	Replaces strong compare and swap operations with weak ones, and calls to set with lazySet
RSK	Removes the synchronized keyword from a method
RSB	Removes a synchronized block
RVK	Removes the volatile keyword
RFU	Removes the finally around an unlock
SKCR	Shrinks a critical region
EXCR	Expands a critical region
SPCR	Splits a critical region into two regions
MVSB	Moves a statement out of a synchronized block
RTXC	Removes a thread call (wait, join, yield, etc.)
RCXC	Removes call to method in concurrency object (locks, semaphores, etc.)
MXC	Modifies permit count in semaphore and modify thread count in latches and barriers
RAT[b]	Replaces the atomic primitive type of a shared variable with an equivalent non-atomic primitive

[a] The mutation operator incorporates significant extensions that are novel.
[b] The mutation operator is novel to this work.

`java.util.concurrency` package, has been expanded to handle virtually every such method, including `compareAndSet`, `addAndGet`, and `incrementAndAdd`.

Our tool works by determining which mutations can be applied to an input program, then randomly selecting a number of mutation operators to apply and locations in the source code on which to apply them. The tool can then be run as many times as needed to generate the desired number of variants. For the purposes of generating benchmarks for performance, we collected a selection of nine different data structure implementations from *The Art of Multiprocessor Programming* by Herlihy and Shavit [29]. This was done for several reasons. (1) Data structures are a frequent target for the application of model checking, (2) textbook examples of data structures tend to explore many different implementations of the same data structure using different concurrency mechanisms (giving rise to wildly different state spaces), and (3) taking baseline programs from the same source made it easier to formulate a one-size-fits-all test harness.

4.3. Experiments

All experiments were performed on an Intel Core 2 quad-core with an Intel Q9550 multi-processor, using version 6 of the JPF, version 7 of the Java runtime environment, and version 3.6 of WEKA. 8 gigabytes of RAM were made available to the Java virtual machine, along with an additional 8 gigabytes of disk space for virtual paging.

To gather data to construct the partitioning of threads, we run the model-checker once, searching the state space by beam search (with a width of one) in a depth-first fashion, stopping once we have observed one complete execution of the program. We note that when using our k-means-based heuristic `TICI`, $k = \max(\lceil\sqrt{n/2}\rceil, 3)$. A common rule of thumb is that $k \approx \sqrt{n/2}$, where n here is the number of threads. Here we pick $k = \max(\lceil\sqrt{n/2}\rceil, 3)$, based on the rule and the fact that all of the benchmarks have one master thread and two classes of worker threads. In some situations, this choice may not produce the optimal value of k, but it would be disingenuous to present results that require prior knowledge that we do not have in the real world.

For each data structure program, we generated three unique versions with different randomly chosen mutations induced by our mutation analysis framework. Those that manifested a bug were kept and used for the performance tests.

For the performance experiments, we collected the execution time and states explored of each heuristic. Each of the benchmarks was run with 4, 6, and 8 worker threads, and we varied the number of producer threads and consumer threads

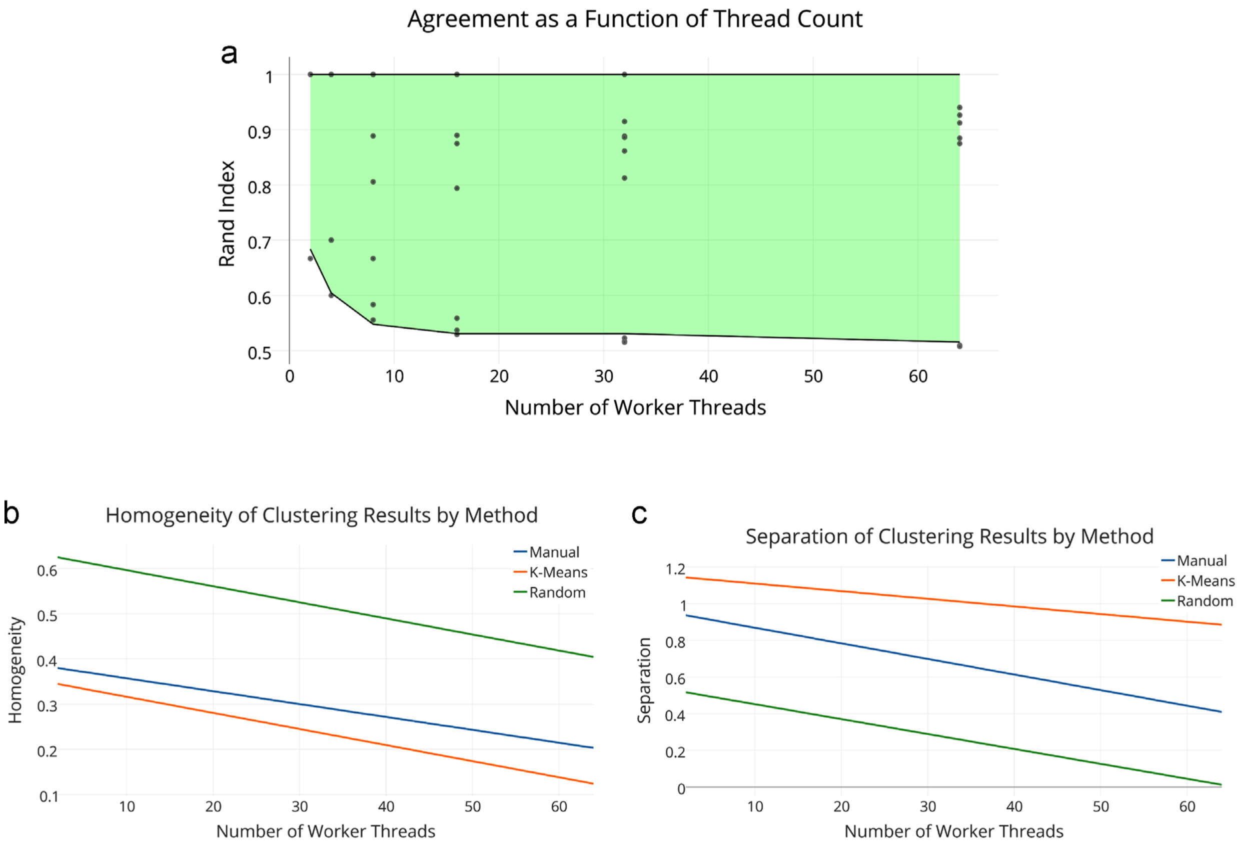

Fig. 5. Results of the agreement and quality tests. In (a), the line at $y = 1$ indicates perfect correspondence of the k-means approach to the manual approach, the curve below indicates the correspondence of random cluster assignments to the manual approach, and the scattered points are the results of the experiments. In (b) and (c), we see linear trends of homogeneity and separation as a function of the number of worker threads.

with the restriction that there were at least two of each (e.g. 4 producers and 4 consumers, 5 producers and 3 consumers, etc.) For the sake of clarity, results are reported separately for each combination of benchmark, thread count, and class distributions. Each experiment was run three times and averages were taken in order to get precise timing results. For all experiments, the model-checker was limited to generating up to 2^{16} states. The recent history limit for both versions of `TICI` and `TI` was set at 10, as all heuristics performed at their best at or around this value.

4.4. Results

4.4.1. Agreement and quality

First, we shall examine the results of the agreement and quality tests as depicted in Fig. 5.

According to the Rand index, the manual and k-means approaches produce extremely similar clustering results. On average, the Rand index between the two approaches is 0.82 (that is, 82% agreement), achieving an index of 1.0 in 32% of experiments and scoring a 0.90 or above in 42% of experiments. The index between the manual approach and random assignment converges to 0.50 as the number of threads increases, and on average the k-means approach generously exceeds this lower bound, which indicates that there is a statistically significant relationship the abstract information gleaned from manual analysis of the source code and the read–write patterns taken from execution traces. However, three of the benchmarks converge to an index of 0.50 as the number of threads increases, which indicates that our automatic approach diverges significantly from the manual approach in these cases:

- **losenotify**: In this benchmark, threads attempt to communicate with one another over a shared object by calling `wait` or `notify` methods repeatedly. They do not do any meaningful work other than this, which means that the only reads and writes we observe are from threads checking and incrementing a local loop index variable in an identical way.
- **nestedmonitor**: The implementation of the buffer is hollow in that no values are actually stored or removed. The only changes made to the data structure are to two semaphores which provide the illusion that the buffer is full or empty depending on the number of times that the `get` and `put` methods were called. Both producers and consumers use the same semaphores and in much the same way, which causes the read–write sequences of each thread to appear the same.
- **wronglock**: The code that each worker thread executes is virtually identical except for the specific locks that they seek to acquire. Since our approach tracks frequencies of reads and writes and not the specific values that threads handle, we cannot identify the different classes of threads.

From the perspective of our manual analysis of the source code, we can easily distinguish the worker threads according to the tasks that they perform, but because the executions of these tasks are extremely similar, our ability to extract useful information from the read–write patterns is limited in these cases. Finally, we observe that the k-means approach performs better than the manual approach both in terms of homogeneity and separation. This is to be expected, because the manual clustering results were not computed according to the read–write patterns. However, the k-means approach performs almost as well as the manual approach and much better than random chance, which again confirms our hypothesis.

4.4.2. Benchmark generation

We will now comment upon the results of the benchmark generation process. Out of the 27 benchmarks generated by our approach, 13 presented concurrency bugs that were detectable using the model checker, a success rate of 48%; see Table 2 for a list of these mutants. Considering the number of different thread counts and thread distributions, this gives us 118 different test cases to use. The outcome was acceptable in that it provided us with enough benchmarks on which to do performance tests, but we identified several ways in which the effectiveness of the approach could be improved.

Table 2
The list of benchmarks generated using mutation analysis along with the class(es) of bugs present in the program. Note that CoarseList(3) is unique in that it has no two-thread bugs, but does have a three-thread bug.

Benchmark	Mutation(s)	Producer–producer bug	Producer–consumer bug	Consumer–consumer bug
CoarseList(1)	SKCR	False	True	False
CoarseList(2)	SKCR	False	False	True
CoarseList(3)	SPCR	False	False	False
LazyList(1)	RFU	False	True	True
LazyList(2)	SKCR	False	True	True
LazyList(3)	SPCR	True	True	True
OptimisticList(1)	RFU	True	True	True
OptimisticList(2)	SKCR	True	True	True
OptimisticList(3)	SPCR	True	True	True
LFQueueRecycle(1)	EAN	False	False	True
LFQueueRecycle(2)	EAN	False	False	True
LFQueueRecycle(3)	WAO	False	False	True
BoundedQueue(1)	SPCR, EAN	False	False	True

First, applying mutations to an input program is not guaranteed to induce a concurrency violation. Some mutation operators may fail to cause a violation, such as an `ESP` (parameter exchange) mutation between two synchronization blocks where the locks are unique to each block. Others have the potential to cancel each other out, such as `SKCR` (shrinking a critical region) followed by `EXCR` (expanding a critical region). However, misapplication of mutations could be avoided by introducing inexpensive checks prior to their introduction; for example, before applying the `ESP` mutation, one could check to see whether there are any additional uses of the parameters of the synchronization blocks, as this would indicate the potential for the mutation to cause havoc.

Second, some mutations can be too coarse-grained to generate a violation. In particular we call to mind the `SPCR` operator: it can split a critical section into two disjoint ones, but if most or all of the shared memory operations are placed underneath an `if`, `while`, or `for` block, then they all get moved together, obviating a violation. It may be necessary to pair such operators with semantics-preserving control flow mutations, such as converting "`if(f(A)) { B; C; }`" to "`X = f(A); if(X) B; if(X) C;`".

Third, we are happy to report that the extension of the `EAN` operator and the introduction the `WAO` operator were helpful in producing usable benchmarks. In particular, there are frequent opportunities to break up calls to atomic functions like `addAndGet`, but in order to generate transformations, it was necessary to have access to type information. That is more easily done when working with a framework for mature languages like ROSE rather than a framework for language prototyping like TXL.

Finally, we see that a mutation on a single piece of code can sometimes induce bugs belonging to multiple classes. When the mutation is applied to a common piece of code like a helper function used to navigate a data structure, then all threads may be affected. Other times, a mutation may introduce a race condition on a shared memory location that one class of threads only reads from and that another class of threads reads from and writes to. This presents a challenge when evaluating model checking heuristics: it is possible that one heuristic may appear better than another on a given benchmark, but it may be the case that each is making optimal decisions yet moving towards different violations at different depths in the state space.

We note that there is an imbalance in the benchmarks that we used for testing. As can be seen in the table, 92% of the benchmarks contain bugs that occur between threads of the same functional class, and only 51% of benchmarks involve bugs that occur between different classes of threads, and these two categories overlap in all cases except for one, CoarseList(1). According to our hypotheses, this should give the h^{-}_{TICI} heuristic an advantage over the h^{+}_{TICI} heuristic. This happens to be useful because the heuristics are symmetric and we get to see the range of performance outcomes that either is capable of.

4.4.3. Performance

A full listing of the results of the performance experiments can be found in the appendix. The highlights of the results of the performance evaluation experiments are as follows:

- In **95%** of tests, either h^{-}_{TICI} or h^{+}_{TICI} outperformed h_{TI} in terms of the number of states explored to reveal a violation. This confirms our claim that at least one version h_{TICI} will (almost always) do better than h_{TI} alone.
- However, when we break down that figure, we observe an asymmetry: h^{-}_{TICI} outperforms h_{TI} 95% of the time, but h^{+}_{TICI} only does so 41% of the time. Moreover, h^{+}_{TICI} performs better than h_{TI} if and only if h^{-}_{TICI} performs better. This is expected due to the imbalance in the benchmarks with respect to the classes of bugs.
- If we examine the hit rate, that is, the frequency with which each technique succeeds in finding a bug under the resource bounds we have set, we see that h_{TI} finds the bug 41% of the time, h^{-}_{TICI} **100%**, and h^{+}_{TICI} 37%.
- In Fig. 7, we break down the success rate further by comparing that rate under increasingly restrictive resource bounds. We find that while making picking the wrong coefficient for h_{TICI} can negatively impact performance, the impact on the quality of the results is very minor. On the other hand, the rewards for making correct choice outweigh the risks.
- Across all benchmarks, on average, h_{TI} locates a violation after exploring 7166 states, compared to h^{-}_{TICI} which does so after exploring 1758 states and h^{+}_{TICI}, after 6897 states. That gives h^{-}_{TICI} a 33% lead on h_{TI}, and h^{+}_{TICI} a 3% disadvantage. However, that is only if we choose to be extremely generous and ignore instances where a heuristic failed to find a bug within the limit. If we include the failures, using the number of states explored as a conservative estimate on the actual number needed to find the bug, we see that h^{-}_{TICI} comes out on top with 1758 states explored, followed by h_{TI} with 14001, and then h^{+}_{TICI} with 27,144. By that measurement, h^{-}_{TICI} finds bugs **7.96 times faster** than h_{TI}, and h^{+}_{TICI} 1.93 times slower.
- Across all combinations of benchmarks and thread counts, analyzing the execution trace and performing the cluster analysis using WEKA took 96 ms on average, with a median training time of 88 ms. With respect to search time, every millisecond spent producing the thread behavior model **saved an average of 5337 ms** on the search when using the h^{-}_{TICI} heuristic. For the h^{+}_{TICI} heuristic, which failed to put the information to good use, lost an extra 624 ms for each millisecond of training.

From the performance data (as can be seen in Fig. 6), it is abundantly clear that the heuristic values reported by h_{CI} are strongly correlated with the distance to a bug, and that the combination of h_{CI} with h_{TI} can be a fruitful marriage. However, we must be cautious with the results because of the issue with the sign of the correlation being unknown. If the selection of programs was flipped with respect to the types of bugs involved, then h^{+}_{TICI} and h^{-}_{TICI} would swap places in terms of the rankings – an 8 × speed-up would become a 2 × slow-down and vice versa. In practice, if one version of the h_{TICI} fails to find a bug within a resource bound, it is likely that re-running the model checker with the other version will succeed. Similarly, it is also possible to run more than one instance of the model checker in parallel. Alternatively, it is possible to employ additional analyses or meta-heuristics to select the best coefficients for h_{TI} and h_{CI}. For example, we recommend using a

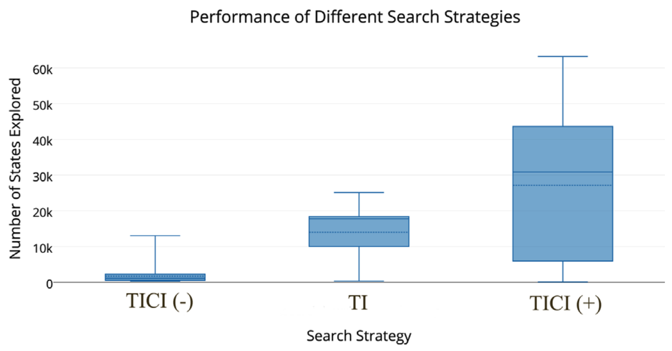

Fig. 6. A box plot of the states explored by each search strategy to find a bug in the 118 different problem instances. The graph illustrates both the tremendous potential of incorporating data from external analyses into the model checking procedure as well as the perils involved in selecting the correct parameters.

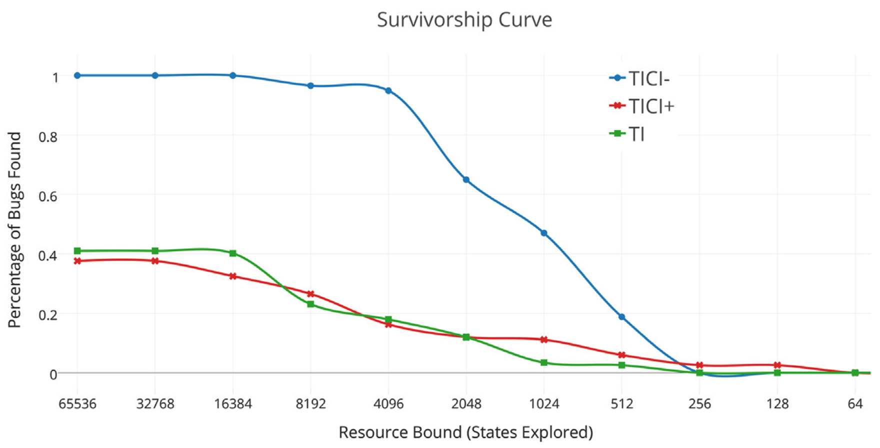

Fig. 7. Survivorship curve for the different search strategies that we investigated. Picking a poor coefficient for h_{TICI} can negatively impact performance, but we are still about as likely to find a bug with h_{TICI} as we are with h_{TI} under the same resource bounds.

linear estimation-of-distribution algorithm as defined by Staunton and Clark [48], or using static analysis to make a list of bug candidates to determine what classes of bugs are most likely present in the program. However, a treatment of that subject goes beyond the scope of this paper.

4.4.4. Threats to validity

While we see our results as cause for celebration, we must also take them with a grain of salt. We have identified several potential threats to the validity of our work that we must address.

First, on the subject of internal validity, the results of the evaluation of our heuristic on mutated programs are lopsided, making it difficult to verify our twin hypotheses. By weakening synchronization constraints, our mutation analysis tends to expose memory accesses to all threads indiscriminately, meaning that producer–consumer bugs are often introduced alongside producer–producer and consumer–consumer bugs. Having multiple classes of bugs occurring at different depths in the state space complicates our interpretation of the results.

Second, because the mutation analysis is unguided, we suffer from a very high rate of experimental mortality: many of our mutated programs failed to contain a detectable bug. While we have reason to believe that our artificial bugs resemble real-world ones, we did not determine how many buggy variants of our input programs were possible. As a result, we cannot say for certain how representative our test cases were of the set of all possible programs.

Next, there are threats to the external validity of our work, that is, the generalizability of our techniques and our results.

First, our clustering technique only applies to a particular class of programs, programs in which we have both a division of labor and the potential for interference between and within classes of threads. Data structures, which are ubiquitous, often fall into this category. However, for certain canonical concurrency problems like the Dining Philosopher's problem, our technique provides no benefit. More to the point, our approach fails when the “communication topology” between threads cannot be linearly divided into self-similar groups.

Second, our prior analysis is inexpensive because we expect the thread behaviors to be separable in a coarse-grained way; we only require a small number of traces and a very low-order model of the read–write patterns of threads. But as we turn our attention to more complex varieties of concurrency bugs, we will no doubt require more and more expensive data. This means that a rigorous cost–benefit analysis will become increasingly important.

Third, we describe our work as a “refinement” of an existing heuristic. This means that the behavior and performance of our novel heuristic are bounded by the original. In our case, the refinement translated to a geometric reduction in search time for problems that the original heuristic was designed to handle. However, that reduction is merely forestalling the inevitable when dealing with exponential growth in the state space; we have beaten back the darkness, but we have yet to expel it.

Finally, our approach is not completely self-contained because we require outside information to select a positive or negative coefficient for our heuristic. Whether that outside information comes from the programmer or from another automated analysis, we are left with deferred costs that we did not consider in our evaluation. In our work, we were able to reduce our dependency to a single bit of information, but we believe that this problem will become more pressing as we develop more sophisticated analyses; accounting for this invisible debt will be crucial to the practical value of such a heuristic.

All these things having been said, we believe our shortcomings indicate that much work has yet to be done in this domain.

5. Related work

Rungta and Mercer [42] present a structural heuristic that uses static analysis to model calling contexts of functions to improve upon a finite state machine distance heuristic. Pelánek et al. [39] demonstrate a simple but effective structural heuristic that values states according to the number of reads and writes that occur in each state's recent history.

In the same vein as structural heuristics, several researchers have proposed a variety of property-independent metaheuristics for explicit-state model-checking. Alba and Troya [1] and Godefroid and Kurshid [24] use genetic algorithms to selectively inject noise into the heuristic search process to flush out interesting paths and thus more efficiently identify concurrency bugs. Seppi et al. [46] apply a Bayesian model that improves upon the values provided by a heuristic according to a confidence estimate based on the distribution of values produced by that heuristic. Chicano and Alba [7] use ant colony optimization techniques alongside partial-order reduction.

Many of the techniques applied in this paper are common in works involving statistical debugging and program comprehension. For instance, Bowring et al. [5,4] present a method based for automatic classification of program behaviors. Aggregate statistical measures are derived from execution data and used to construct Markov models. These Markov models are labeled according to the behaviors that are exhibited (e.g. buggy vs. clean, light-workload vs. heavy-workload), and are then used to train classifiers to help predict and detect behaviors in future executions. The quality of these classifiers is improved through a process of active learning: the software system iteratively collects new executions of the program, attempts to classify the resulting data, and queries the programmer/designer to test its accuracy. This is a supervised learning task, which presupposes that we have samples of the behaviors we interested in (in our case, property violations) and relies on extensive human interaction to produce good results. Our approach differs in that ours is an unsupervised learning task; we merely seek to identify similarities in the behavioral patterns of different threads without regard to the program properties we would like to verify. Furthermore, we allow the model-checker to do all of the heavy lifting, which minimizes the need for human intervention.

6. Conclusion and future work

In this work, we have presented a framework for mining execution traces of concurrent programs for read–write patterns and a novel heuristic to exploit models trained on these patterns. We extended a well-established structural heuristic for locating race conditions described in Groce and Visser [27], which seeks to schedule as many different threads as frequently as possible, with an extension that helps ensure the fair treatment of different groups or cohorts of threads. To divide threads into functional categories (e.g. producers and consumers), execution traces were collected and mined for read–write patterns, expressed as Markov chain models. K-means clustering was performed on the set of these models, assigning each thread to a group. Our experiments, which relied in part upon a novel application of a mutation analysis framework, have shown that our refined heuristic can outperform its conventional counterpart; this is especially interesting considering that the models were trained on relatively few inputs (just one execution trace). Our hope is that this work can serve as a model for future explorations into the combination of model checking with other analyses.

While we see the performance results as cause to celebrate, we also understand that a geometric reduction in search time is merely forestalling the inevitable; we have beaten back the darkness, but we have yet to expel it. For our future work, we intend on pairing model checking with neural program analysis that provides linear time approximations of expensive static analyses. Our preliminary results have shown that, in some cases, this combination allows us to achieve *optimal* search time results at arbitrary thread counts. In tandem with this research, we are also investigating strategies for combining distributed search with metaheuristics, in order to automatically parallelize the search process when we encounter a choice between incommensurate alternatives as we have seen in this paper.

Appendix A. Appendix

See Table A1.

Table A1
Data gathered during performance testing. Note that time is reported in milliseconds.

Benchmark [producers, consumers]	States explored (TI)	Search time (TI)	Bug found (TI)	States explored (TICI(−))	Search time (TICI (−))	Bug found (TICI(−))	States explored (TICI(+))	Search time (TICI (+))	Bug found (TICI(+))	Training time
BoundedQueue(1)[2,2]	12,339	8810	True	3486	2692	True	17,368	115,797	True	110
BoundedQueue(1)[2,4]	20,607	426,402	False	6096	78,354	True	29,711	351,794	False	94
BoundedQueue(1)[2,6]	18,403	583,674	False	6388	717,167	True	810	894	True	87
BoundedQueue(1)[3,3]	20,440	1,692,962	False	3931	157,157	True	23,727	404,255	False	98
BoundedQueue(1)[3,5]	18,203	682,375	False	2602	137,606	True	44,529	107,912	False	93
BoundedQueue(1)[4,2]	19,827	905,665	False	10,011	521,763	True	29,305	555,991	False	82
BoundedQueue(1)[4,4]	18,203	625,603	False	13,040	210,415	True	35,076	216,898	False	91
BoundedQueue(1)[5,3]	18,203	668,937	False	12,381	542,126	True	44,529	146,620	False	102
BoundedQueue(1)[6,2]	18,203	1,099,031	False	8684	182,320	True	54,555	409,603	False	89
CoarseList(1)[2,2]	2039	4131	True	302	443	True	4981	4059	True	572
CoarseList(1)[2,4]	338	1731	True	302	863	True	37,047	45,865	False	91
CoarseList(1)[2,6]	17,851	347,190	False	302	2458	True	43,641	477,897	False	82
CoarseList(1)[3,3]	25,117	578,083	False	302	1186	True	6917	138,052	True	79
CoarseList(1)[3,5]	16,195	503,080	False	302	2898	True	36,534	1,269,522	False	80
CoarseList(1)[4,2]	9997	174,749	True	302	307	True	14,019	179,373	True	82
CoarseList(1)[4,4]	16,747	55,190	False	302	1336	True	35,076	44,942	False	88
CoarseList(1)[5,3]	14,404	241,814	True	302	791	True	44,529	400,554	False	79
CoarseList(1)[6,2]	17,407	436,537	False	302	1460	True	54,555	1,501,443	False	79
CoarseList(2)[2,2]	2677	3283	True	2477	1386	True	4063	2226	True	200
CoarseList(2)[2,4]	1631	3051	True	1327	1612	True	31,387	68,183	False	87
CoarseList(2)[2,6]	18,402	283,893	False	1327	4470	True	43,641	1,304,429	False	83
CoarseList(2)[3,3]	21,110	1,467,323	True	1533	50,601	True	17,880	509,215	True	86
CoarseList(2)[3,5]	18,402	390,948	False	1327	8247	True	36,534	636,366	False	81
CoarseList(2)[4,2]	22,975	1,747,566	False	2477	162,474	True	31,799	254,591	False	75
CoarseList(2)[4,4]	18,402	64,886	False	1327	4027	True	35,076	43,369	False	82
CoarseList(2)[5,3]	18,402	947,608	False	1533	42,904	True	44,529	296,198	False	81
CoarseList(2)[6,2]	18,402	477,051	False	2477	1760	True	54,555	586,384	False	82
CoarseList(3)[2,2]	10,394	10,496	True	2778	1669	True	15,024	11,320	True	95
CoarseList(3)[2,4]	22,531	678,024	False	1520	1468	True	30,783	527,979	False	102
CoarseList(3)[2,6]	17,773	410,324	False	1520	1808	True	43,641	755,451	False	84
CoarseList(3)[3,3]	22,008	502,372	False	1726	1853	True	26,849	346,664	False	78
CoarseList(3)[3,5]	17,773	285,630	False	1520	1516	True	36,534	513,838	False	81
CoarseList(3)[4,2]	21,996	550,652	False	2778	3311	True	31,524	711,480	False	83
CoarseList(3)[4,4]	17,773	256,982	False	1520	400,230	True	35,076	387,361	False	81
CoarseList(3)[5,3]	17,773	325,397	False	1726	249,433	True	44,529	616,146	False	81
CoarseList(3)[6,2]	17,773	626,370	False	2778	128,208	True	54,515	811,327	False	81
LazyList(1)[2,2]	3824	5047	True	520	459	True	518	463	True	144
LazyList(1)[2,4]	1117	2999	True	520	1776	True	3053	4652	True	101
LazyList(1)[2,6]	18,403	516,003	False	520	1626	True	8757	321,183	True	84
LazyList(1)[3,3]	1117	4502	True	520	657	True	2635	2047	True	87
LazyList(1)[3,5]	18,403	475,367	False	539	1394	True	35,076	567,269	False	85
LazyList(1)[4,2]	1117	129,129	True	1013	1381	True	51,815	231,032	False	81
LazyList(1)[4,4]	18,403	63,060	False	539	784	True	44,529	240,515	False	86
LazyList(1)[5,3]	18,403	546,168	False	539	20,413	True	54,555	334,757	False	88
LazyList(1)[6,2]	18,403	1,150,406	False	1013	2553	True	55,021	453,535	False	88
LazyList(2)[2,2]	486	445	True	402	410	True	105	182	True	183
LazyList(2)[2,4]	340	540	True	402	506	True	105	159	True	82
LazyList(2)[2,6]	18,403	449,758	False	402	1091	True	105	745	True	72

LazyList(2)[3,3]	1109	1584	True	402	922	True	303	281	True	88
LazyList(2)[3,5]	18,403	703,921	False	402	617	True	303	534	True	76
LazyList(2)[4,2]	2889	74,069	True	402	498	True	571	672	True	81
LazyList(2)[4,4]	18,403	81,715	False	402	731	True	571	463	True	80
LazyList(2)[5,3]	18,403	1,321,258	False	402	559	True	572	517	True	93
LazyList(2)[6,2]	18,403	946,077	False	402	548	True	584	974	True	88
LazyList(3)[2,2]	522	490	True	544	534	True	2400	1402	True	159
LazyList(3)[2,4]	4140	4970	True	544	493	True	30,251	63,584	False	98
LazyList(3)[2,6]	18,403	773,327	False	544	2237	True	43,641	209,642	False	90
LazyList(3)[3,3]	4140	4290	True	544	29,628	True	11,532	541,417	True	86
LazyList(3)[3,5]	18,403	834,170	False	544	1,009,415	True	36,534	1,133,228	False	89
LazyList(3)[4,2]	4140	16,533	True	544	701	True	33,189	591,499	False	86
LazyList(3)[4,4]	18,403	282,820	False	544	4776	True	35,076	202,003	False	90
LazyList(3)[5,3]	18,403	747,438	False	544	1121	True	44,529	730,606	False	90
LazyList(3)[6,2]	18,403	1,536,713	False	544	32,969	True	54,555	583,256	False	91
LFQueueRecycle(1)[2,2]	6378	5626	True	2096	1035	True	5606	4643	True	98
LFQueueRecycle(1)[2,4]	2423	34,480	True	2449	1598	True	26,300	45,781	True	92
LFQueueRecycle(1)[2,6]	13,255	434,181	True	3119	2305	True	43,641	886,456	False	83
LFQueueRecycle(1)[3,3]	10,079	729,413	True	2302	17,261	True	4428	52,425	True	91
LFQueueRecycle(1)[3,5]	13,987	279,102	True	2838	2291	True	36,534	343,469	False	84
LFQueueRecycle(1)[4,2]	20,983	1,244,991	False	2086	2023	True	31,862	320,102	False	76
LFQueueRecycle(1)[4,4]	14,092	505,119	True	2748	50,749	True	35,076	869,680	False	92
LFQueueRecycle(1)[5,3]	15,492	2,335,505	False	2388	77,587	True	44,529	588,172	False	84
LFQueueRecycle(1)[6,2]	15,546	838,060	False	2170	823,887	True	54,555	524,135	False	82
LFQueueRecycle(2)[2,2]	6220	14,381	True	2096	1012	True	5470	3407	True	91
LFQueueRecycle(2)[2,4]	2423	10,384	True	2609	1609	True	26,270	371,379	True	89
LFQueueRecycle(2)[2,6]	13,255	310,640	True	3339	175,539	True	43,641	622,184	False	82
LFQueueRecycle(2)[3,3]	10,117	110,954	True	2302	1595	True	4428	33,939	True	79
LFQueueRecycle(2)[3,5]	13,987	453,258	True	2998	82,671	True	36,534	298,997	False	83
LFQueueRecycle(2)[4,2]	21,232	745,431	False	2086	3724	True	32,012	2,605,639	False	75
LFQueueRecycle(2)[4,4]	14,092	494,732	True	2748	2273	True	35,076	466,235	False	83
LFQueueRecycle(2)[5,3]	15,401	1,829,008	True	2388	53,890	True	44,529	460,334	False	84
LFQueueRecycle(2)[6,2]	15,597	642,855	False	2170	11,302	True	54,555	805,498	False	82
LFQueueRecycle(3)[2,2]	6219	4090	True	2096	1014	True	5470	3339	True	88
LFQueueRecycle(3)[2,4]	2423	6142	True	2449	1526	True	26,270	279,498	True	87
LFQueueRecycle(3)[2,6]	13,255	185,619	True	3119	80,904	True	43,641	1,034,928	False	82
LFQueueRecycle(3)[3,3]	10,117	72,167	True	2302	2784	True	4428	3388	True	80
LFQueueRecycle(3)[3,5]	13,987	182,481	True	952	711	True	55,021	494,258	False	83
LFQueueRecycle(3)[4,2]	21,232	562,869	False	2086	157,830	True	32,012	1,462,141	False	76
LFQueueRecycle(3)[4,4]	14,092	494,235	True	2748	2506	True	35,076	904,415	False	83
LFQueueRecycle(3)[5,3]	15,401	2,109,415	True	2388	141,480	True	44,529	415,044	False	83
LFQueueRecycle(3)[6,2]	15,597	967,969	False	2170	1609	True	54,555	665,017	False	82
OptimisticList(1)[2,2]	2751	2033	True	477	418	True	405	349	True	114
OptimisticList(1)[2,4]	20,285	369,248	False	477	814	True	3314	72,411	True	98
OptimisticList(1)[2,6]	18,392	416,107	False	477	2157	True	10,901	854,558	True	98
OptimisticList(1)[3,3]	20,023	4,549,608	False	771	21,474	True	4596	509,153	True	87
OptimisticList(1)[3,5]	18,403	239,829	False	771	26,514	True	25,987	249,076	False	87
OptimisticList(1)[4,2]	1377	12,828	True	771	997	True	12,532	117,086	True	88
OptimisticList(1)[4,4]	18,403	190,248	False	771	844	True	24,285	905,816	False	88
OptimisticList(1)[5,3]	18,403	419,590	False	717	1033	True	25,879	774,329	False	91
OptimisticList(1)[6,2]	18,403	613,510	False	717	1490	True	23,866	1,510,040	False	93
OptimisticList(2)[2,2]	1801	1228	True	452	357	True	388	323	True	115
OptimisticList(2)[2,4]	1378	8929	True	787	655	True	6065	7458	True	105
OptimisticList(2)[2,6]	18,403	257,124	False	787	780	True	15,541	229,774	True	92

Table A1 (*continued*)

OptimisticList(2)[3,3]	15,701	654,240	True	769	247,277	True	4755	3799	True	144
OptimisticList(2)[3,5]	18,403	944,612	False	769	7105	True	4802	129,271	True	96
OptimisticList(2)[4,2]	1382	2596	True	717	774	True	1088	1116	True	94
OptimisticList(2)[4,4]	18,403	319,009	False	717	905	True	28,601	295,123	False	89
OptimisticList(2)[5,3]	18,403	311,888	False	717	963	True	25,879	316,256	False	91
OptimisticList(2)[6,2]	18,403	1,060,525	False	717	5255	True	23,866	675,986	False	91
OptimisticList(3)[2,2]	8512	5640	True	1294	1568	True	17,250	8882	True	106
OptimisticList(3)[2,4]	20,285	65,053	False	1310	1036	True	35,176	140,228	False	106
OptimisticList(3)[2,6]	18,392	1,552,430	False	1310	216,606	True	42,976	204,678	False	92
OptimisticList(3)[3,3]	20,010	824,431	False	1194	1921	True	46,873	138,862	False	99
OptimisticList(3)[3,5]	18,403	254,264	False	1194	1322	True	63,214	190,377	False	94
OptimisticList(3)[4,2]	20,087	751,594	False	1142	19,704	True	30,887	426,292	False	88
OptimisticList(3)[4,4]	18,403	218,612	False	1142	1165	True	43,493	200,850	False	97
OptimisticList(3)[5,3]	18,403	1,403,409	False	1142	8434	True	29,404	316,328	False	100
OptimisticList(3)[6,2]	18,403	1,478,386	False	1142	1954	True	32,055	276,242	False	101

References

[1] Alba E, Troya JM. Genetic algorithms for protocol validation. In: Parallel problem solving from nature—PPSN IV. Springer; 1996. p. 869–79.
[2] Andrews JH, Briand LC, Labiche Y. Is mutation an appropriate tool for testing experiments? In: Proceedings of the 27th international conference on software engineering. ACM; 2005. p. 402–11.
[3] Bhadra J, Abadir MS, Wang LC, Ray S. A survey of hybrid techniques for functional verification. IEEE Des Test Comput 2007;(2):112–22.
[4] Bowring JF, Harrold MJ, Rehg JM. Improving the classification of software behaviors using ensembles of control-flow and data-flow classifiers. Technical report GIT-CERCS-05-10. Georgia Institute of Technology; 2005.
[5] Bowring JF, Rehg JM, Harrold MJ. Active learning for automatic classification of software behavior. In: ACM SIGSOFT software engineering notes, vol. 29. ACM; 2004. p. 195–205.
[6] Bradbury JS, Cordy JR, Dingel J. Mutation operators for concurrent java (j2se 5.0). In: Second workshop on mutation analysis, 2006. IEEE; 2006. p. 11.
[7] Chicano F, Alba E. Ant colony optimization with partial order reduction for discovering safety property violations in concurrent models. Inf Process Lett 2008;106(6):221–31.
[8] Cordy JR. Txl—a language for programming language tools and applications. In: Electronic notes in theoretical computer science, vol. 110; 2004. p. 3–31.
[9] Cordy JR. The txl source transformation language. Sci Comput Program 2006;61(3):190–210.
[10] Cordy JR, Dean TR, Malton AJ, Schneider KA. Source transformation in software engineering using the txl transformation system. Inf Softw Technol 2002;44(13):827–37.
[11] Cordy JR, Halpern-Hamu CD, Promislow E. Txl: a rapid prototyping system for programming language dialects. Comput Lang 1991;16(1):97–107.
[12] Delahaye M, Du Bousquet L. A comparison of mutation analysis tools for java. In: 2013 13th international conference on quality software (QSIC). IEEE; 2013. p. 187–95.
[13] Delamaro M, Pezze M, Vincenzi AM, Maldonado JC. Mutant operators for testing concurrent java programs. In: Brazilian symposium on software engineering; 2001. p. 272–85.
[14] Détienne F. Expert programming knowledge: a schema-based approach. arXiv preprint cs/0702003; 2007.
[15] Dickinson W, Leon D, Podgurski A. Finding failures by cluster analysis of execution profiles. In: Proceedings of the 23rd international conference on software engineering. IEEE Computer Society; 2001. p. 339–48.
[16] Do H, Elbaum S, Rothermel G. Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact. Empir Softw Eng 2005;10(4):405–35.
[17] Do H, Rothermel G. A controlled experiment assessing test case prioritization techniques via mutation faults. In: Proceedings of the 21st IEEE international conference on software maintenance, 2005. ICSM'05. IEEE; 2005. p. 411–20.
[18] El-Ramly M, Stroulia E, Sorenson P. Recovering software requirements from system-user interaction traces. In: Proceedings of the 14th international conference on software engineering and knowledge engineering. ACM; 2002. p. 447–54.
[19] Engler D, Musuvathi M. Static analysis versus software model checking for bug finding. In: Verification, model checking, and abstract interpretation. Springer; 2004. p. 191–210.
[20] Eytani Y, Ur S. Compiling a benchmark of documented multi-threaded bugs. In: Proceedings of 18th international parallel and distributed processing symposium, 2004. IEEE; 2004. p. 266.
[21] Farchi E, Nir Y, Ur S. Concurrent bug patterns and how to test them. In: Proceedings of international parallel and distributed processing symposium, 2003. IEEE; 2003. p. 7.
[22] Flanagan C, Godefroid P. Dynamic partial-order reduction for model checking software. In: ACM sigplan notices, vol. 40. ACM; 2005. p. 110–21.
[23] Fu Q, Lou JG, Wang Y, Li J. Execution anomaly detection in distributed systems through unstructured log analysis. In: Ninth IEEE international conference on data mining, 2009. ICDM'09. IEEE; 2009. p. 149–58.
[24] Godefroid P, Khurshid S. Exploring very large state spaces using genetic algorithms. In: Tools and algorithms for the construction and analysis of systems. Springer; 2002. p. 266–80.
[25] Godefroid P, Nagappan N. Concurrency at microsoft: an exploratory survey. In: CAV workshop on exploiting concurrency efficiently and correctly; 2008.
[26] Groce A, Visser W. Model checking java programs using structural heuristics. In: ACM SIGSOFT software engineering notes, vol. 27. ACM; 2002. p. 12–21.
[27] Groce A, Visser W. Heuristics for model checking java programs. Int J Softw Tools Technol Transf 2004;6(4):260–76.
[28] Hall M, Frank E, Holmes G, Pfahringer B, Reutemann P, Witten IH. The weka data mining software: an update. ACM SIGKDD Explor Newslett 2009;11(1):10–8.
[29] Herlihy M, Shavit N. The art of multiprocessor programming, Revised Reprint. Elsevier; 2012.
[30] Jia Y, Harman M. An analysis and survey of the development of mutation testing. IEEE Trans Softw Eng 2011;37(5):649–78.
[31] Jiang L, Su Z. Context-aware statistical debugging: from bug predictors to faulty control flow paths. In: Proceedings of the twenty-second IEEE/ACM international conference on automated software engineering. ACM; 2007. p. 184–93.
[32] Kaufman L, Rousseeuw PJ. Finding groups in data: an introduction to cluster analysis, vol. 344. John Wiley & Sons; 2009.
[33] Kuhn A, Ducasse S, Girba T. Semantic clustering: identifying topics in source code. Inf Softw Technol 2007;49(3):230–43.
[34] Lloyd S. Least squares quantization in pcm. IEEE Trans Inf Theory 1982;28(2):129–37.
[35] Lu S, Park S, Seo E, Zhou Y. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: Proceedings of the 13th international conference on architectural support for programming languages and operating systems. ASPLOS XIII. New York, NY, USA: ACM; 2008. p. 329–39. ⟨http://doi.acm.org/10.1145/1346281.1346323⟩.
[36] Magee J, Kramer J. State models and java programs. Wiley; 1999.
[37] Mancoridis S, Mitchell BS, Rorres C, Chen Y, Gansner ER. Using automatic clustering to produce high-level system organizations of source code. In: International conference on program comprehension. IEEE Computer Society; 1998. p. 45.
[38] Păsăreanu CS, Dwyer MB, Visser W. Finding feasible abstract counter-examples. Int J Softw Tools Technol Transf 2003;5(1):34–48.
[39] Pelánek R, Rosecky` V, Moravec P. Complementarity of error detection techniques. In: Electronic notes in theoretical computer science, vol. 220, no. 2; 2008. p. 51–65.
[40] Quinlan D. Rose: compiler support for object-oriented frameworks. Parallel Process Lett 2000;10(02n03):215–26.
[41] Rand WM. Objective criteria for the evaluation of clustering methods. J Am Stat Assoc 1971;66(336):846–50.
[42] Rungta N, Mercer EG. A context-sensitive structural heuristic for guided search model checking. In: Proceedings of the 20th IEEE/ACM international conference on automated software engineering. ACM; 2005. p. 410–3.
[43] Rungta N, Mercer EG. Hardness for explicit state software model checking benchmarks. In: Fifth IEEE international conference on Software engineering and formal methods, 2007. SEFM 2007. IEEE; 2007. p. 247–56.
[44] Safyallah H, Sartipi K. Dynamic analysis of software systems using execution pattern mining. In: 14th IEEE international conference on program comprehension, 2006. ICPC 2006. IEEE; 2006. p. 84–88.
[45] Schordan M, Quinlan D. A source-to-source architecture for user-defined optimizations. In: JMLC'03: joint modular languages conference. Lecture notes in computer science, vol. 2789. Springer-Verlag; August 2003. p. 214–23.
[46] Seppi K, Jones M, Lamborn P. Guided model checking with a Bayesian meta-heuristic. Fundam Inf 2006;70(1):111–26.
[47] Sistla AP. Employing symmetry reductions in model checking. Comput Lang Syst Struct 2004;30(3):99–137.

[48] Staunton J, Clark JA. Finding short counterexamples in promela models using estimation of distribution algorithms. In: Proceedings of the 13th annual conference on genetic and evolutionary computation. ACM; 2011. p. 1923–30.
[49] Visser W, Păsăreanu CS, Khurshid S. Test input generation with java pathfinder. In: ACM SIGSOFT software engineering notes, vol. 29, no. 4; 2004. p. 97–107.
[50] Welles O, Kodar O. F for fake; 1974.
[51] Wolpert DH, Macready WG. No free lunch theorems for search. Technical report. Technical report SFI-TR-95-02-010. Santa Fe Institute; 1995.
[52] Wu LL, Kaiser GE. Constructing subtle concurrency bugs using synchronization-centric second-order mutation operators; 2011.