

(A Work-in-progress Paper)

A Novel Test Coverage Metric for Concurrently-Accessed Software Components

Serdar Tasiran, Tayfun Elmas, Guven Bolukbasi, and M. Erkan Keremoglu

Koç University, Istanbul, Turkey

{stasiran,telmas,gbolukbasi,mkeremoglu}@ku.edu.tr

Abstract. We propose a novel, practical coverage metric called “location pairs” (LP) for concurrently-accessed software components. The LP metric captures well common concurrency errors that lead to atomicity or refinement violations. We describe a software tool for measuring LP coverage and outline an inexpensive application of predicate abstraction and model checking for ruling out infeasible coverage targets.

1 Introduction

Verification and testing of concurrently-accessed software components is particularly challenging because the interleaving of concurrently executed threads compounds the program state space. Validation methods must both store information and have control of thread scheduling. As a result, exhaustive testing or model checking are prohibitively costly for realistic configurations of industrial-scale concurrent programs. This motivates the study of methods that combine verification and testing approaches in order strike a compromise between computational cost and exhaustiveness.

We are exploring several hybrid techniques in which coverage metrics serve as the link between model checking and testing tools and enable us to (i) quantify the adequacy testing/verification performed, (ii) communicate partial results and testing/verification goals between tools, (iii) direct testing effort towards unexplored, quantitatively distinct executions of a program that are interesting for a particular purpose.

In this paper, we propose a coverage metric, called “location pairs” (LP) focused on the concurrency aspects of test executions. The LP metric was inspired by a pattern common to atomicity and refinement violations that were found in industrial software examples and in the literature [2, 3, 5]. These concurrency errors are triggered whenever an instance of the following scenario takes place: A thread t_1 executes a particular line of code ln_1 , then a context-switch occurs and another thread t_2 starts execution at another line of code ln_2 . This sequence of events, and the existence of a dependency between t_1 ’s execution of the block of code preceding ln_1 and t_2 ’s execution of the block of code following ln_2 guarantee that the error will occur regardless of the program state and other concurrent threads. This scenario is different from a race condition: t_1 and t_2 may protect all global variables they access using the proper locks, but the interleaving described may still cause an atomicity or refinement violation. The LP metric is associated with higher-level concurrency errors similar to those in [1, 2, 5].

The fact that the LP metric seems to correspond well with certain types of concurrency errors makes it a promising tool for guiding manual validation effort and catching errors beyond the reach of methods based on state-space traversal. A particular concurrency error may not be possible with small program states or few threads because certain conflicts or resource contention are required to trigger it. It may also be unlikely because it requires the external events to be timed and threads interleaved a certain way. The LP metric captures errors caused by such unforeseen interleavings. The oversight typically occurs because of erroneous assumptions about the environment or the belief that a synchronization mechanism makes a certain interleaving impossible. Given an unexercised but apparently reachable location pair as a target, the programmer can reason about conditions that need to be set up to reach the target or provide the coverage analysis tool hints that help it prove that the target location pair is unreachable. In this way, the LP metric helps test writers explore qualitatively distinct and error-prone scenarios. Further, if they believe a certain scenario is not possible, it provides a tool for them to make explicit and check the justification for their belief.

The use of the LP metric also makes possible practical automatic techniques to be provided for the tasks of measuring coverage, ruling out infeasible coverage targets, and providing abstract traces to help with test input generation. We are developing a software tool that measures test coverage according to the LP metric for Java programs. We also propose the lightweight use of formal verification tools (a combination of predicate abstraction and model checking) to rule out an efficiently-computable set of unreachable location pairs. By avoiding exhaustive exploration of possible program states and thread interleavings and precise determination of reachable location pairs, we keep the computational burden of our method low. Since the metric involves consideration of pairs of method bodies, it does not lead to a combinatorial blow-up in the number of coverage targets.

Section 2 presents preliminaries required to state our coverage metric precisely. A motivating example from Java class libraries is provided in Section 3. Section 4 describes the LP coverage metric. Section 5 outlines the method used to compute a reduced set of coverage targets. Section 6 describes how the proposed metric successfully captures the concurrency errors in the examples studied. Section 7 outlines our future research.

2 Preliminaries

2.1 Concurrent Programs: Syntax

In this paper, we focus on realizations of concurrently accessible data structures written in object-oriented languages. For ease of exposition, we use a simple language that we call CJ. The syntax of CJ is given in Fig. 1. A data structure \mathcal{D} is an instantiation of a CJ class \mathcal{C} . The set of \mathcal{C} 's methods are denoted by $\mathcal{M}_{\mathcal{C}}$.

`atomic(s)` marks s as an *atomic* statement block: a sequence of actions, which can be shown to be atomic by using commutativity and reduction arguments. `pure(s)` marks s as a *pure* statement block as defined in [4]. Roughly speaking, a pure block does not modify the program state unless it terminates exceptionally. The concepts of purity and atomicity are used in defining the LP metric in order to reduce the number of qualitatively distinct scenarios that need to be explored.

$P ::= \text{defn}^* e$	(program)
$\text{defn} ::= \text{class } cn \text{ body}$	(class declaration)
$\text{body} ::= \text{extends } c \{ \text{field}^* \text{ method}^* \}$	(class body)
$\text{field} ::= t \text{ fn} = e$	(field declaration)
$\text{method} ::= t \text{ mn}(\text{arg}^*) \{e\}$	(method declaration)
$\text{arg} ::= t x$	(variable declaration)
$s, t ::= c \mid \text{int} \mid \text{boolean} \mid \dots$	(type)
$c ::= cn \mid \text{Object}$	(class type)
$e ::= \text{new } c$	(allocate*)
$ x$	(variable*)
$ e.f$	(field access*)
$ e.f := e$	(field assignment*)
$ e == e \mid e < e$	(comparison expression*)
$ e \vee e \mid e \wedge e \mid \neg e$	(boolean expression*)
$ e.mn(e^*)$	(method call)
$ \text{synchronized } e \text{ in } e$	(synchronization)
$ \text{fork } e$	(fork)
$ \text{if}(\text{Bexp}?s : s)$	(if statement)
$ \text{while}(\text{Bexp}, s)$	(while statement)
$ \text{pure}(s)$	(purity annotation)
$ \text{atomic}(s)$	(atomicity annotation)
$cn \in \text{ClassNames}$	
$fn \in \text{FieldNames}$	
$mn \in \text{MethodNames}$	
$x, y \in \text{VariableNames}$	

Fig. 1. The grammar for CJ

2.2 Concurrent Programs: Semantics

The semantics of a data structure written as a CJ class is given by a state transition graph. Each *state* of the program is a unique assignment of values to program variables. *Global variables* \mathcal{V}_G are variables in the representation of \mathcal{D} that are accessible by all methods of \mathcal{C} . Each multi-threaded execution also uniquely defines a set of *local variables* \mathcal{V}_L which are variables accessible by an individual thread only. Local variables correspond to method-local variables in CJ.

Each state transition corresponds to an atomic update of a program variable, called an *action*. For ease of exposition, we assume that the types of expressions shown with an asterisk (*) in Fig. 1 are executed atomically as well as method invocations, returns, lock acquisitions and releases.

Control Flow Graphs: With each method $\mu \in \mathcal{M}$ we associate a control flow graph CFG_μ obtained from the CJ code for the method. A control flow graph $CFG_\mu = \langle V, E, \lambda_v, \lambda_e \rangle$ is a directed graph. Vertices of a CFG are partitioned into two: $V = V_{ctrl} \cup V_{exec}$, where V_{ctrl} is the set of *branching vertices* corresponding to “if” and “while” statements, and V_{exec} is the set of *execution vertices*. Each branching vertex $v \in V_{ctrl}$ is labeled with a single atomically-evaluated expression $\lambda_v(v)$. The two outgoing edges representing the two branches are labeled by the corresponding boolean value of $\lambda_v(v)$. Each execution vertex $v \in V_{exec}$ is labeled by a sequence of actions $\lambda_e(v)$.

A *location* in the CFG_μ is like a program counter – it indicates at what point of the code the execution is. More precisely, a CFG_μ has associated with it a set of locations L_μ where each $l \in L_\mu$ is identified by either a pair of actions (α_i, α_{i+1}) where α_i and α_{i+1} are two consecutive actions in the label of a vertex in the CFG,

or represents the entry point of a vertex in the CFG and corresponds to the case where execution of actions labeling that vertex has not started yet. The action immediately following a location l is denoted by $\alpha(l)$.

3 Motivating example: java.lang.StringBuffer

The control flow graph for the `append` method of an older version of `StringBuffer` is given in Fig. 2. This version of the `append` implementation has a concurrency error due to the fact that the method argument, `sb`, is not locked throughout the method [5].

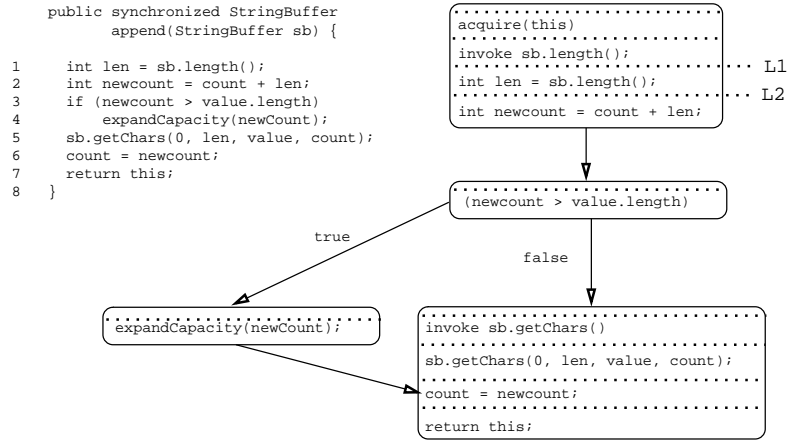


Fig. 2. The CFG for `StringBuffer.append()`

During an execution of `append` by a thread t_1 , the state of `sb` can be modified by another thread t_2 between the invocation of the (synchronized) method `sb.length()` in line 1 of the `append` method and the invocation of `sb.getChars()` in line 5. For example, if `sb`'s length is more than 0 and t_2 executes `sb.setLength(0)` between these two invocations, an atomicity violation occurs. `setLength(0)` invokes the following line

```
count = newLength;
```

where the value of `newLength` is 0. Let “L3 \rightarrow L4” denote this action in `setLength()`. Since this action occurs before `sb`'s contents are appended to `this`, in fact, an exception is thrown during `getChars`.

Observe that this error occurs whenever line 1 of `append` is followed by the line `count = newLength;` in `sb.setLength`. We found that this pattern of two particular consecutive dependent actions from two method bodies explains all atomicity and refinement violations we encountered. The LP metric makes precise and captures this intuition in the form of a coverage metric. Since the metric requires pairwise consideration of method bodies, it does not lead to a combinatorial blow-up in the number of coverage targets.

4 The “Location Pairs” Coverage Metric

We formulate our coverage metric as a requirement that certain states and certain transitions of a set of coverage finite-state machines (FSM) be traversed during test executions of the program. A coverage FSM $\mathcal{F}^{1,2}$ is defined for each pair

of methods (μ_1, μ_2) . A fragment of the coverage FSM for with μ_1 chosen to be `StringBuffer.append()` and $\mu_2 = \text{setLength}()$ is given in Fig. 3. The $L3 \rightarrow L4$ transition represents the `lineCount = newLength` in `setLength()`. The states and transitions of the coverage FSM $\mathcal{F}^{1,2}$ are described below.

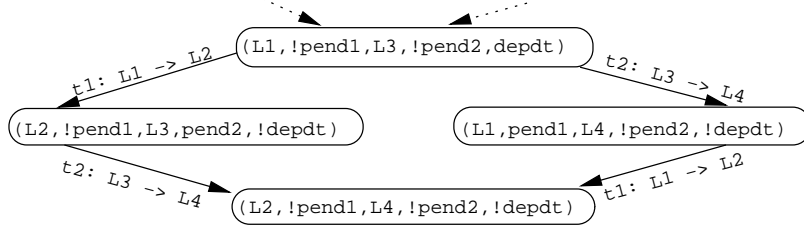


Fig. 3. A fragment of the coverage state machine for a concurrent execution of `StringBuffer.append()` and `StringBuffer.setLength()`

A state s of $\mathcal{F}^{1,2}$ is a tuple $s = \langle l^1, \text{pend}^1, l^2, \text{pend}^2, \text{depdt} \rangle$ where

- l^1 and l^2 are locations in the CFG's of μ_1 and μ_2 , respectively,
- `depdt` is a boolean variable indicating whether the pair of actions $\alpha(l^1)$ and $\alpha(l^2)$ immediately following locations l^1 and l^2 are *dependent* as defined in [9], i.e., at least one of them modifies a variable that the other one reads or writes a different value to, and
- `pend1` (respectively, `pend2`) is a boolean variable that has the value `true` iff the current coverage state was reached by taking an action in μ_1 (respectively, μ_2) from a previous coverage state where `depdt` was `true`.

There is a transition in the coverage FSM whenever one location can be followed by another. This will be reduced later to approximate what can happen. There is a transition in the coverage FSM from state p to state q

$$p = \langle l_p^1, \text{pend}_p^1, l_p^2, \text{pend}_p^2, \text{depdt}_p \rangle \longrightarrow q = \langle l_q^1, \text{pend}_q^1, l_q^2, \text{pend}_q^2, \text{depdt}_q \rangle$$

iff `depdtq` is a legal value for whether actions $\alpha(l_q^1)$ and $\alpha(l_q^2)$ are dependent, and one of the followings holds:

- (i) the execution of method μ_1 can move from l_p^1 to l_q^1 , and $l_p^2 = l_q^2$, or
- (ii) the execution of method μ_2 can move from l_p^2 to l_q^2 , and $l_p^1 = l_q^1$.

The `pend1` and `pend2` bits are updated as follows:

- If `depdtp` is `false` then both `pendq1` and `pendq2` are assigned to `false`.
- If `depdtp` is `true` and case (i) above was applied, then `pendq2` is assigned to `true` and `pendq1` is assigned to `false`.
- If `depdtp` is `true` and case (ii) above was applied, then `pendq1` is assigned to `true` and `pendq2` is assigned to `false`.

Our coverage metric requires that all reachable transitions of the coverage FSM of the following two forms be traversed:

$$p = \langle l_p^1, \text{true}, l^2, \text{pend}_p^2, \text{depdt}_p \rangle \xrightarrow{\alpha(l^1)} q = \langle l_q^1, \text{pend}_q^1, l^2, \text{pend}_q^2, \text{depdt}_q \rangle$$

$$p = \langle l^1, \text{pend}_p^1, l_p^2, \text{true}, \text{depdt}_p \rangle \xrightarrow{\alpha(l^2)} q = \langle l^1, \text{pend}_q^1, l_q^2, \text{pend}_q^2, \text{depdt}_q \rangle$$

During an execution of a multi-threaded program, several different pairs of threads can be in the process of executing μ_1 and μ_2 concurrently. Each such pair of threads is at a particular, possibly different state of the coverage FSM $\mathcal{F}^{1,2}$.

The following reductions are used to obtain a more compact CFG and fewer locations from a method without eliminating any interesting interleavings:

- We lump together a basic block (i.e. uninterrupted by branching statements) of *method-local actions* – actions that do not modify any global variables, and consider it as a single action. Any method-local action is independent from any other action by another thread, therefore, its interleavings do not produce qualitatively distinct scenarios.
- A statement block s marked as “pure” (`pure(s)`) is partitioned into a control flow graph of atomic actions as described above. A pure execution of such a block is interpreted as no action having been taken at all, since a pure execution does not modify any global variables (see [4]). This results in an important reduction in the number of target interleavings, since pure executions typically perform only lock acquisitions and releases and without this optimization, they can lead to a large number of possible, equivalent interleavings.
- During coverage analysis, we model atomic executions of lock-protected blocks marked `atomic($\alpha_s; \dots; \alpha_t$)` as a pair of consecutive actions ($\alpha^{\llbracket}, \alpha^{\rrbracket}$). α^{\llbracket} is an aggregate action that acquires all the locks in the beginning of the atomic block and α^{\rrbracket} releases the locks acquired previously as well as achieving the composed effect of $\alpha_s, \dots, \alpha_t$. This is done in order to incorporate into the model a possible violation of the claimed atomicity: If a thread gets interleaved in between the pair of actions and modifies a global variable that was supposed to have been protected by the locks of the atomic block, an error is signaled.

Even though applying the reductions above yields a smaller, more usable coverage FSM, an exact determination of the reachable set of states and transitions of $\mathcal{F}^{1,2}$ remains undecidable. To have a practical method, we instead employ an inexpensive technique that uses predicate abstraction and model checking to rule out a subset of unreachable states and the transitions of $\mathcal{F}^{1,2}$. Due to space limitations, a detailed description is deferred to the Appendix.

5 Measuring coverage

While we make conservative simplifying assumptions while reducing the coverage FSM, during actual coverage measurement, no such approximation is needed. Whether a pair of actions executed one after the other are dependent can be easily and exactly determined at run-time by examining the types and parameters of the actions by the coverage tool.

If at any point in the execution of a multi-threaded program, some thread \mathbf{t}_1 starts executing μ_1 while another thread \mathbf{t}_2 is executing μ_2 (or \mathbf{t}_2 starts executing μ_2 while \mathbf{t}_1 is executing μ_1), the coverage tool creates a new instance $\mathcal{F}_i^{(1,2)}$ of the class representing the coverage FSM $\mathcal{F}^{1,2}$ and starts it at a state corresponding to the pair of locations that \mathbf{t}_1 and \mathbf{t}_2 are in. From then on $\mathcal{F}_i^{(1,2)}$ takes transitions triggered by the actions of \mathbf{t}_1 and \mathbf{t}_2 as described in Section 4 until either \mathbf{t}_1 exits that particular execution of μ_1 or \mathbf{t}_2 exits μ_2 , whichever comes earlier. The coverage tool keeps an instance $\mathcal{F}_{rec}^{(1,2)}$ of the coverage FSM $\mathcal{F}^{1,2}$ for record keeping. All edges

of $\mathcal{F}^{1,2}$ visited by any instance of $\mathcal{F}^{1,2}$ created during execution are recorded in $\mathcal{F}_{rec}^{(1,2)}$. Note that different edges of $\mathcal{F}^{1,2}$ may be covered by different pairs of threads.

The feedback provided to the programmer after running a test suite is a list of unexercised pairs of locations that the analysis described above has not been able to rule out as a possibility. At this point, the programmer can either identify this as a true coverage gap and write test programs aimed at exercising those scenarios, or, if he believes this is an unreachable pair of locations, he can provide his reasoning in the form of additional predicates to the coverage feasibility analysis in order to rule out the fictitious coverage gap (see the Appendix). Note that the programmer has to provide his reasoning rather than simply turning off the warning from the coverage tool.

```

1 public synchronized void addElement(Object obj) {
2     modCount++;
3     ensureCapacityHelper(elementCount + 1);
4     elementData[elementCount] = obj;
5     elementCount++;
6 }

1 public int lastIndexOf(Object elem) {
2     int count = Read(elementCount) - 1;
3     return lastIndexOf(elem, count);
4 }

```

Fig. 4. Code fragments illustrating the error in `java.util.Vector`

6 Empirical Evidence for The Metric

This section describes how the LP coverage metric captures concurrency errors from the literature and errors in industrial examples we studied. Each error scenario as described is easily expressed as one of the required edges for the coverage FSM for the pair of methods referred to for each example.

java.util.StringBuffer: This example, the concurrency error associated with it and how the location pairs metric captures it were discussed in Section 3.

java.util.Vector: The code for this example is modified to contain one atomic action per line for illustration purposes (see Fig. 4). If line 3 in `lastIndexOf` is followed by line 5 in line 4 in `addElement`, this leads to an atomicity violation, which was previously discovered by [6]. In particular, if `elem == obj`, this would have led to an incorrect return value for `lastIndexOf`, which is a refinement violation [2].

Cache Module of Boxwood: The error, explained in more detail in [2], involves a cache block in a data structure to be flushed to the next level of the storage hierarchy while it is being overwritten by another thread. This corresponds to line 3 in the `CpToCache` method in Fig. 5 being executed right after line 3 in the `Flush` method, representing flush midway through the copying of buffer `buf` to the cache.

The “Scan” file system: The error in this system, as documented in [10], is very similar in spirit to the scenario in the Boxwood cache. While the file system cache is being written to the disk, after a block gets copied to disk and gets marked “clean”, it gets overwritten by another file system thread.

The concurrency error categories in [3] The LP metric can express as a coverage goal all error-prone scenarios that are described in this work. The errors in the category “Code Assumed to Be Protected” of [3] are particularly relevant for atomicity and refinement violations.

7 Ongoing Work

We are implementing a software tool written in Java that measures LP metric coverage attained during testing. We instrument the byte-code of the program

```

1 private static void CpToCache(byte[] buf,          1 public static void Flush(int lsn) {
    CacheEntry te, int lsn, Handle h) {           ...
2   for(int i = 0; i < buf.Length; i++) {         2 lock(clean) {
3     te.data[i] = buf[i];                        3   BoxMain.alloc.Write(h, te.data, te.data.Length,
4   }                                              0, 0, WRITE_TYPE.RAW);
5   te.lsn = lsn;                                4 }
6 }                                              ...

```

Fig. 5. Buggy code fragment from an earlier version of the Boxwood Cache module under test by inserting notification calls from the tested program to the coverage tool after each code block interpreted as an atomic action as described above. To minimize impact of online coverage analysis on the concurrency behavior of the original program, in a later version of the tool, we intend to instrument the program being tested in order to generate per-thread logs of actions relevant to the coverage metric. The coverage tool will then take only the logs as its input and will not affect the execution of the program being tested.

We also intend to study bug databases of other large multi-threaded designs, such as web servers, and determine to what extent the proposed metric captures the bugs documented. Future research includes an implementation of the coverage FSM reduction technique described in the Appendix.

References

1. C. Artho, K. Havelund, and A. Biere. High-Level Data Races. In VVEIS '03, Int'l Workshop on Verification and Validation of Enterprise Information Systems.
2. T. Elmas, S. Tasiran, and S. Qadeer. VyrD: Verifying concurrent programs by runtime refinement-violation detection. In *Proc. ACM SIGPLAN 2005 Conf. on Programming Language Design and Implementation, PLDI 2005*. ACM Press, June 2005.
3. Eitan Farchi, Yarden Nir, Shmuel Ur. Concurrent Bug Patterns and How to Test Them. 17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France, page 286.
4. C. Flanagan, S. Freund, and S. Qadeer. Exploiting purity for atomicity. In *Proc. Intl. Symposium on Software Testing and Analysis (ISSTA 2004)*. ACM Press, 2004.
5. C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multi-threaded programs. In *Proc. 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 256–267, 2004.
6. C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN 2003 Conf. on Programming Language Design and Implementation, PLDI '03*.
7. B. Long, P. Strooper, and L. Wildman. A method for verifying concurrent java components based on an analysis of concurrency failures. *Concurrency and Computation: Practice and Experience*, 00(1-7):1–13, 2005.
8. B. Long and P. A. Strooper. A classification of concurrency failures in java components. 17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France., page 287.
9. P. Godefroid. Partial order methods for the verification of concurrent systems—an approach to the state-explosion problem. In *Lecture Notes in Computer Science*, volume 1032. Springer-Verlag, 1996.
10. S. Tasiran, A. Bogdanov, and M. Ji. Detecting concurrency errors in file systems by runtime refinement checking. Tech. Report HPL-2004-177, HP Labs, 2004.
11. R. N. Taylor, D. L. Levine, and C. D. Kelly. Structural testing of concurrent programs. *IEEE Trans. Softw. Eng.*, 18(3):206–215, 1992.

Appendix

Reduction of The Coverage FSM Using Predicate Abstraction and Model Checking

Because of synchronization mechanisms used and the conditionals in branching expressions certain states of the coverage FSM, are not reachable. For the coverage tool to be practical, as many of these infeasible coverage FSM states and transitions need to be ruled out as is efficiently possible using automated analysis. For each coverage FSM $\mathcal{F}^{1,2}$, a set of predicates $\text{Preds}^{1,2}$ is defined consisting of the following predicates:

- For each lock corresponding to a synchronization expression *synchronized e in e'* appearing in the method body for μ_1 or μ_2 outside of a pure block, a boolean variable $\text{lockheld}(\mathbf{e}, \mathbf{t}_i)$ indicating whether or not thread \mathbf{t}_i is holding lock $\ell(\mathbf{e})$.
- For each boolean expression \mathbf{e} that occurs as the branching condition in an **if** or **while** statement, a boolean variable \mathbf{p}_e if and only if the following conditions are satisfied: the locks for all global variables referred to by \mathbf{e} or that influence any of the local variables referred to by \mathbf{e} are held by the thread executing the conditional statement.
- For each local variable ν_L^1 in μ_1 and ν_L^2 in μ_2 , a boolean variable $\text{equal}(\nu_L^1, \nu_L^2)$

A coverage boolean program is obtained using the predicate abstraction of the concrete program using the predicates above. A model checker is used that at each step, non-deterministically, executes an action from the boolean program obtained from μ_1 , the one obtained from μ_2 , or a boolean program representing the environment. The environment model encapsulates the conservative assumption that any non-lock global variable that is not protected by a lock along μ_1 or μ_2 can be changed arbitrarily by the multi-threaded environment. A reachability analysis of this boolean program by the model checker gives us the reduced FSM $\mathcal{F}^{1,2}$.

To keep reachability analysis conservative, whenever aliasing cannot be ruled out as a cause of actions α_1 in μ_1 and α_2 in μ_2 being dependent, α_1 and α_2 are considered to be possibly dependent and the corresponding states and transitions are not removed from the coverage FSM.

Intuitively, this analysis rules out location pairs that can be determined to be infeasible by only doing local analysis on the pair of method bodies without making extra assumptions about the program state, which can be included by the programmer as an option. The purpose of this choice is to keep the analysis simple and efficient. The rationale is that the programmer has followed the same approach to convince herself of the correctness of her method implementations.

Related Work

Extensive research has been performed on test adequacy criteria for sequential programs and coverage measurement and improvement tools have been developed. However, there is relatively little work on coverage metrics geared towards concurrency errors and measuring how well different interleavings of a multi-threaded program have been explored. [11] defines a hierarchy of structural coverage metrics for testing multi-threaded programs. However, the complexity of the metrics increase exponentially with the number of concurrent threads and makes them impractical for industrial-scale systems.

Long et. al. [8,7] propose the use of Petri-net models for representing Java synchronization primitives. Firings of these nets correspond to different actions that can be performed with the primitive and are correlated with likely low-level errors such as race conditions. Method control flow graphs are annotated with firings of the net and test sets are constructed manually in order to achieve coverage of the firings. Their analysis is focused on a single synchronized object, which requires a separate application for each shared variable in a concurrently-accessed component – a fact that makes their approach impractical for our purposes. Farchi et. al. [3] provide a classification of concurrency errors and testing heuristics that make it likelier that these errors will be exercised, but do not provide a way of measuring when and how well testing has addressed possible errors of this sort.

Model checking techniques make use of partial order techniques to reduce the number of interleavings of concurrent threads that need to be explored [9]. Roughly speaking, a pair of actions executed by two different threads are declared independent unless one writes to a shared variable that the other reads from or writes to. Commuting a pair of such actions is said to result in an equivalent execution and only one execution from an equivalence class defined in this way is explored during model checking. The set of equivalence classes that this give rise to could be seen as a coverage metric for concurrent programs. However, since the coverage state induced by this metric includes both data, control and the interleaving aspects of a program, it is prohibitively complex and does not result in a usable coverage metric for large, practical programs.