A Study of Interleaving Coverage Criteria

Shan Lu, Weihang Jiang and Yuanyuan Zhou
Department of Computer Science,
University of Illinois at Urbana Champaign, Urbana, IL 61801
{shanlu,wjiang3,yyzhou@cs.uiuc.edu}

ABSTRACT

Concurrency bugs are becoming increasingly important due to the prevalence of concurrent programs. A fundamental problem of concurrent program bug detection and testing is that the interleaving space is too large to be thoroughly explored. Practical yet effective interleaving coverage criteria are desired to systematically explore the interleaving space and effectively expose concurrency bugs.

This paper proposes a concurrent program interleaving coverage criteria hierarchy, including **seven** (including **five new**) coverage criteria. These criteria are all designed based on different concurrency fault models. Their cost ranges from exponential to linear.

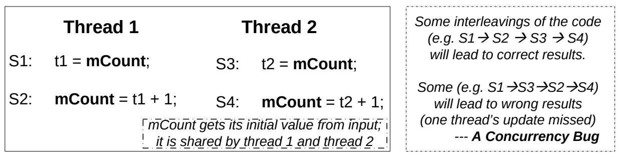
Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging–Testing tools

General Terms: Reliability

Keywords: interleaving, coverage criteria, concurrent program

1. INTRODUCTION

Concurrent programs, specifically multi-threaded and multi-processed programs on shared memory machines, have become increasingly important in the past few years. Unfortunately, concurrent programs are *prone* to bugs due to the inherent complexity of concurrency. Even worse, the software testing and bug detection efforts [4, 10] to improve the concurrent program quality are greatly obstructed by the problematic *interleaving space* of the concurrent programs.



```
          Thread 1                      Thread 2
S1:   t1 = mCount;               S3:   t2 = mCount;

S2:   mCount = t1 + 1;           S4:   mCount = t2 + 1;
```

mCount gets its initial value from input; it is shared by thread 1 and thread 2

Some interleavings of the code (e.g. S1→S2 →S3 →S4) will lead to correct results.

Some (e.g. S1→S3→S2→S4) will lead to wrong results (one thread's update missed)
*--- **A Concurrency Bug***

Figure 1: An example simplified from a real Mozilla data race bug. The selected interleaving affects the final value of `mCount` and whether the bug is revealed.

Interleaving is an important and unique domain of concurrent programs. Unlike the execution of sequential programs, which is almost solely determined by the input, the execution and bug triggering of concurrent programs are greatly affected by the non-deterministic interleaving among concurrent execution components (threads or processes). As shown in Figure 1, *one* program input could generate different results due to different interleaving orders. In order to expose a concurrency bug, *we need to explore not only different inputs but also different interleavings for one input*, as tried in real world and previous research [4].

Unfortunately, real world concurrent programs have huge interleaving spaces (factorial to the execution length for each program input). Real world testing resource can only check a small portion of the interleaving space. A lot of concurrency bugs inevitably skip into production runs.

In order to effectively explore the large interleaving space, good *coverage criteria* are required. Statement [1] and data flow coverage [3, 6] are widely used to help select *representative inputs*. Similarly, interleaving coverage criteria is needed to help *select representative interleavings* to effectively test concurrent programs and expose concurrency bugs.

Existing interleaving coverage criteria [2, 5, 11] are quite limited. Many of them are either too complicated, with exponential cost, or not based on solid concurrency fault models. Furthermore, there is no wide-spectrum interleaving coverage criteria *hierarchy* yet. In order to provide software testers more choices and researchers better understanding of the trade-offs between coverage criteria's cost and bug-exposing capability, good coverage criteria hierarchies are desired.

This paper makes the following contributions:

(1) A wide-spectrum interleaving coverage criteria hierarchy *Seven* interleaving coverage criteria (*five out of the seven are newly proposed in this paper*) are proposed. They are designed based on different concurrency fault models and represent different levels of software quality requirements. Following subsumption relationship [16], these criteria compose a five-layer hierarchy. This hierarchy can provide a wide spectrum of choices and guidelines to effectively explore the interleaving space to expose concurrency bugs.

(2) Analysis of the cost of the proposed coverage criteria Our cost analysis shows that the proposed coverage criteria range from exponential cost to linear cost. Some of our proposed criteria have linear or quadratic coverage cost and also have solid concurrency fault model basis. They all have the potential to help the practical concurrent program testing.

2. CONCEPTS ON COVERAGE CRITERIA

A program coverage criterion usually focuses on test case selection from a certain program testing space, e.g. the input space, the interleaving space, etc. A criterion C includes two parts. One is a set Γ of program properties, which could be program statements, program branches, etc. One is a property-satisfaction function f, indicating what test cases can satisfy (exercise) a certain program property. The adequacy of a testing is measured by C by checking how many program properties are satisfied (exercised). If all the program properties are satisfied, the testing achieves *complete coverage* and is called a *complete testing* under C.

Cost and *bug-exposing capability* are the two most important metrics for a coverage criterion. A coverage criterion's cost can be measured by the number of test cases needed to exercise all the properties [14]. The capabilities of exposing hidden program bugs are different among testings guided by different coverage criteria, because different coverage criteria have different focus on exercising program properties.

It is usually difficult to reach good balance between cost and bug-exposing capability. That is why people used to compose hierarchical families [1, 6] of coverage criteria in order to gain a thorough understanding of the design tradeoffs. In general, a good criterion should be based on an valid fault models. For example, structural coverage criteria are based on the fault model that most sequential bugs are related to certain program structures and control flows.

3. A MODEL FOR INTERLEAVING

Consider a concurrent program P, executed under an input I, consisting of M threads: $1, 2, ..., M$. Similar to previous work [13], we model the concurrent execution of P by a sequence of shared variable access events. We use E to denote the set of all shared variable accesses, and P_E for a program P with access set E under a given input. At any moment only one thread i is active and executes one event. When i finishes, one thread j (j might be equal to i) will be chosen and executes its next event. The event execution order within each thread is fixed. The order among different threads might change. Each different order to execute P_E is called an **interleaving**. Formally speaking, an *interleaving* $\prec$ of P_E is a total order relation on E. An event e is executed before an event e' iff $e \prec e'$. The whole interleaving domain of P_E is the set of all total order relations on E that maintain the sequential order within each thread.

4. INTERLEAVING COVERAGE CRITERIA

This section presents seven coverage criteria with different interleaving property sets. They are designed based on different concurrency fault models, starting from the most conservative bug assumption—most exhaustive criterion, and ending with the most aggressive (focused) assumption—most simplified criterion. These criteria together build a wide-spectrum hierarchy. Note that among the seven criteria, two of them (criterion 1 and 4.A) have been proposed before, while the other five are newly proposed in this paper.

4.1 Coverage Criteria Description

• **Criterion 1: all-interleavings (ALL)** *The interleaving space gets a "complete coverage" based on ALL, iff all feasible interleavings of shared accesses from all threads are covered (Figure 2(a)).*

We start with a simple and exhaustive interleaving coverage criterion. It is clearly the most expensive yet also the most powerful in exposing concurrency bugs.

Property Set: *ALL* criterion treats every interleaving as one property for testing. The property set size is the total number of possible interleavings and can be calculated as following (N_i is number of access events from thread i):

$$|\Gamma_{\text{ALL}}| = \prod_{i=1}^{M} \binom{\sum_{j=i}^{M} N_j}{N_i}$$

Example: As shown in Figure 2(a), ALL includes all interleavings of the events from the four threads. Its property set size is as large as 63063000 for such a simple program!

• **Criterion 2: thread-pair-interleavings (TPAIR)** *Interleaving space gets 'complete coverage' under TPAIR, if all feasible interleavings of all shared memory accesses from any pair of threads are covered (Figure 2(b)).*

Starting from this model, we begin to use some concurrency fault models to gradually generalize our property set and decrease the coverage cost.

Concurrency Fault Model: Similar to pair-wise testing, this criterion is based on the fault model which assumes that most concurrency bugs are caused by the interaction between two threads, instead of all threads. This is a widely believed concurrency fault model and is used in previous bug detection work [8, 10].

Property Set: TPAIR's property set contains every possible interleaving on all the accesses from every pair of threads (Γ=$\{\prec_{i,j}\}$ $(1 \le i < j \le M)$). A program interleaving $\prec$ on P_E satisfies $\prec_{i,j}$ when the order of all the events from thread i and j in $\prec$ is the same as that in $\prec_{i,j}$.

The property set size can be calculated as following:

$$|\Gamma_{\text{TPAIR}}| = \sum_{1 \le i < j \le M} \binom{N_i + N_j}{N_i}$$

Example: As shown in Figure 2(b), one TPAIR property for thread 1 and 2 could be $\prec_{1,2}$: $r_1^1 \prec_{1,2} r_2^1 \prec_{1,2} w_3^1 \prec_{1,2} w_4^1 \prec_{1,2} r_1^2 \prec_{1,2} w_2^2 \prec_{1,2} r_3^2 \prec_{1,2} w_4^2$. Since this property has no constraint on thread 3 and 4, totally 900900 $(= \binom{16}{8} \cdot \binom{8}{4})$ program interleavings of all four threads can satisfy this property. The whole property set has size: 420.

• **Criterion 3: single-variable-interleavings (SVAR)** *The interleaving space gets a "complete coverage" under criterion SVAR, iff all feasible interleavings of all shared accesses to any specific variable from any pair of threads are covered (Figure 2(c)).*

Concurrency Fault Model: This criterion is based on the observation that many concurrency bugs involve conflicting accesses to *one* shared variable, instead of multiple variables. This is a widely-adopted assumption in many concurrency bug detection, such as the lockset race detection [10].

Property Set: It includes every interleaving of all accesses from two threads to one shared variable ($\Gamma_{\text{SVAR}} = \prec_{i,j,v}$, $1 \le i < j \le M, v \in V$). An interleaving $\prec$ of P_E satisfies property $\prec_{i,j,v}$ iff: all events in thread i and j accessing v have the same order specified in $\prec$ as that in $\prec_{i,j,v}$.

The property set size is as following ($N_{i,v}$ is the number of accesses from thread i to variable v):

$$|\Gamma_{\text{SVAR}}| = \sum_{1 \le i < j \le M} \sum_{v \in V} \binom{N_{i,v} + N_{j,v}}{N_{i,v}}$$

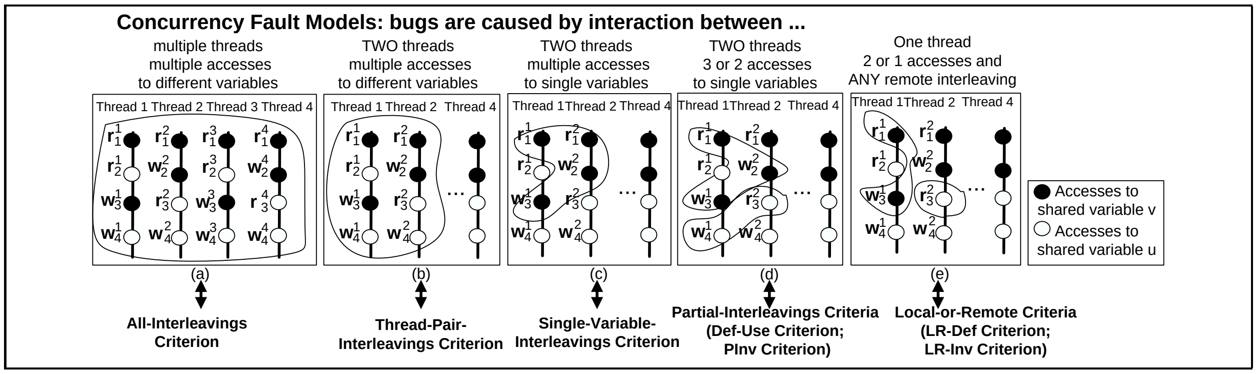

Concurrency Fault Models: bugs are caused by interaction between ...
multiple threads multiple accesses to different variables
TWO threads multiple accesses to different variables
TWO threads multiple accesses to single variables
TWO threads 3 or 2 accesses to single variables
One thread 2 or 1 accesses and ANY remote interleaving
Thread 1 Thread 2 Thread 3 Thread 4
Thread 1 Thread 2 Thread 4
Accesses to shared variable v
Accesses to shared variable u
(a)
(b)
(c)
(d)
(e)
All-Interleavings Criterion
Thread-Pair-Interleavings Criterion
Single-Variable-Interleavings Criterion
Partial-Interleavings Criteria (Def-Use Criterion; PInv Criterion)
Local-or-Remote Criteria (LR-Def Criterion; LR-Inv Criterion)

Figure 2: Different concurrency fault models and corresponding testing coverage criteria (The fault models on the right are more focused and the corresponding testing costs are smaller than those on the left. The solid and hollow circles in the figure represent memory accesses to two different shared variables.)

Example: As shown in Figure 2(c), an SVAR property is an interleaving among accesses to v or u from any pair of threads. For example, one property $\prec_{1,2,v}$ involving thread 1, thread 2 and variable v could be: $r_1^1 \prec_{1,2,v} w_3^1 \prec_{1,2,v} r_1^2 \prec_{1,2,v} w_2^2$. Since this property has no constraint on thread 3, 4 and u access w_4^1, it can be satisfied by totally 4504500 ($= 5 \cdot \binom{16}{8} \cdot \binom{8}{4}$) interleavings. The size of the whole property set Γ_{SVAR} is 72, much smaller than that of Γ_{TPAIR}.

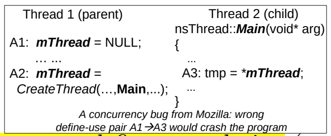
Criterion 4: partial-interleavings (PI) To further reduce the coverage cost, we can consider partial interleavings, i.e. execution scheduling among a small number of accesses, based on the observation that *many concurrency bugs such as data races are caused by wrong order or interaction among two or three accesses, instead of a complete interleaving among all accesses, to one shared variable.*

This fault model is more aggressive than previous ones, yet still quite reasonable and used in previous bug detection [8] and model checking [9]. It can have different variants. Here we discuss two variants as following:

- **Criterion 4.A: define-use (Def-Use)** *The interleaving space gets a "complete coverage" under the Def-Use criterion, iff all possible define-use pairs are covered (Figure 2(d)).*

A similar criterion, a member of the data-flow coverage criteria family [6], is used for sequential programs for decades and was recently extended for concurrent programs [7, 15].

Concurrency Fault Model: The underlying assumption is that, many bugs are caused by a read access using a variable defined by a wrong writer, i.e. wrong define-use relation (an example is shown in the figure on the right).



```
Thread 1 (parent)                Thread 2 (child)
                                 nsThread::Main(void* arg)
A1:  mThread = NULL;             {
     ... ...                       ...
A2:  mThread =                     A3: tmp = *mThread;
  CreateThread(...,Main,...);      ...
                                 }
```

A concurrency bug from Mozilla: wrong define-use pair A1→A3 would crash the program

Property Set: Each property under Define-Use criterion is simply a read-write access pair, where read-access reads the value defined by the write-access. An interleaving $\prec$ satisfies a define-use property $\gamma = (w,\, r)$ iff the write w happens before the read r in the interleaving and there is no other write to the variable read by r between them.

If we use N^r to denote the total number of read accesses in a program, and $N_{i,v}^r$ ($N_{i,v}^w$) for the total number of read (write) accesses in thread i to variable v, the size of the Def-Use property set can be calculated by:

$$|\Gamma_{\text{Def-Use}}| = N^r + \sum_{1 \le i \ne j \le M} \sum_{v \in V} (N_{i,v}^r \cdot N_{j,v}^w)$$

Example: As shown in Figure 2(d), a Def-Use property is a pair of write-read accesses to the same variable. For example, all properties related to access r_3^2 are: (w_4^1, r_3^2), $(\cdot, r_3^2)$, (w_4^3, r_3^2), (w_4^4, r_3^2), where the first element in each pair is the defining write, and $(\cdot, r_3^2)$ means that access r_3^2 reads the initial value in memory. The size of the whole property set $\Gamma_{\text{Def-Use}}$ is 32, smaller than that of Γ_{SVAR}.

- **Criterion 4.B: pair-interleave (PInv)** *The interleaving space gets a "complete coverage" under PInv, if for each consecutive access-pair from any thread, all feasible interleaving accesses to it have been covered (Figure 2(d)).*

In this criterion and the following criterion 5.B, we use *consecutive pair* or *consecutive accesses* to denote a consecutive access pair from one thread accessing the same shared variable (e.g. pair (r_1^1, w_3^1) in Figure 2(d)); we also use *interleaving access* to denote a remote access from another thread. This access reads or writes the same variable between a consecutive pair described above (e.g. w_2^2 could become an interleaving access to above pair in Figure 2).

Concurrency Fault Model: Recent work [8, 12] has shown that unexpected interleaving accesses to a consecutive pair can indicate many concurrency bugs such as atomicity violations (an example is shown in Figure 1). PInv criterion is exactly based on this fault model.

Property Set: Each property is a triplet composed of two consecutive accesses, e_1^i and e_2^i from a thread i, and an interleaving access e^j ($j \neq i$) from another thread j. An interleaving $\prec$ on P_E satisfies a property $\gamma = (e_1^i, e_2^i;\, e^j)$ when the interleaving access e^j occurs in the middle of e_1^i and e_2^i under $\prec$, i.e. $e_1^i \prec e^j \prec e_2^i$.

We use PN to denote the number of all consecutive access pairs in the program; $PN_{i,v}$ for the number of consecutive pairs in thread i on variable v. Specifically, $PN_{i,v} = N_{i,v} - 1$, if $N_{i,v} > 0; otherwise PN_{i,v} = 0$. Then the size of the PInv property set can be calculated by:

$$|\Gamma_{\text{PInv}}| = PN + \sum_{1 \le i \ne j \le M} \sum_{v \in V} (PN_{i,v} \cdot N_{j,v}),$$

Example: As shown in Figure 2(d), a PInv property is a pair of consecutive v (or u) accesses from one thread and an interleaving access from another thread. For example, all properties corresponding to variable v and thread 1, 2 are: $(r_1^1, w_3^1; r_1^2)$ $(r_1^1, w_3^1; w_2^2)$ $(r_1^1, w_3^1; \cdot)$ $(r_1^2, w_2^2;\ r_1^1)$ $(r_1^2, w_2^2; w_3^1)$ $(r_1^2, w_2^2; \cdot)$. The 3rd event in each parentheses is the interleaving access; $\cdot$ means no interleaving. $\Gamma_{\text{Def-Use}}$ has totally 56 properties, less than that of Γ_{SVAR}.

Criterion 5: local-or-remote (LR)

To further reduce the coverage cost, we can use more aggressive fault models. One such direction is to extend criterion 4.A and 4.B to focus on whether there *exists* an interleaving access or a remote variable definer, instead of *all* possible interleaving accesses or *all* possible remote definers. In this way, we only need to consider two properties for each consecutive pair or reader— *whether the remote influence exists or not*. Of course, this relaxation might lead to missing more concurrency bugs. In the following, we briefly discuss the criterion 5.A, extended from criterion 4.A, and 5.B, extended from criterion 4.B.

• **Criterion 5.A: local-or-remote-define (LR-Def)** *Interleaving space gets "complete coverage" under LR-Def, if for each read-access r in the program, both of the following cases have been covered — r reads a variable defined by local thread (or the initial memory state) and r reads a variable defined by a different thread (Figure 2(e)).*

Concurrency Fault Model: The fault model is similar to the general LR criteria described above.

Property Set: The property set $\Gamma_{\text{LR-Def}}$ includes at most two properties for each read access r: r reading a value defined by a local thread's write access; and r reading value defined by a different thread's write.

The property set size ranges between N^r and $2N^r$, where N^r is the number of all reads in P_E.

• **Criterion 5.B: local-or-remote interleave (LR-Inv)** *Interleaving space gets "complete coverage" under LR-Inv, if for every consecutive access pair (e, e'), from any thread accessing any shared variable, both of the following cases have been covered — (e, e') has an unserializable interleaving access and (e, e') does not have one (Figure 2(e)).*

Here, a **(un)serializable interleaving** is an interleaving that has (not) equivalent effects to a serial execution [8, 12].

Concurrency Fault Model: This criterion extends the general criterion 5 in a way similar to criterion 4.B by focusing on interleaving access to consecutive access pair.

Property Set: Similar to criterion 5.A, the property set $\Gamma_{\text{LR-Inv}}$ includes at most two properties for each consecutive access pair: unserializable interleaving case and no interleaving or serializable interleaving case.

The property set size ranges between the number of all possible consecutive pairs, PN, and $2PN$.

Overall, criterion ALL subsumes criterion TPAIR, which subsumes SVAR, which subsumes PI and LR. All together, they build a hierarchy. Due to the space limit, we omit the proof here.

5. COST ANALYSIS

The *cost* (refer to section 2) of the above seven coverage criteria can be approximated based on the property-set size and how many properties an interleaving can cover. As we can see in table 1, ALL, TPAIR and SVAR criteria all require exponential number of interleavings for a complete coverage, too expensive in practice. The other four criteria Def-Use, PInv, LR-Def and LR-Inv are much better, with only quadratic or linear cost. Of course, low cost usually comes at the expense of bug exposing capabilities. Evaluating these criteria's bug exposing capability is remained as our future work.

	ALL	TPAIR	SVAR
Upper-Bound	$O((\frac{4^n}{\sqrt{n}})^{(M-1)})$	$O(\frac{4^n}{\sqrt{n}} \cdot M)$	$O(\frac{4^{n/V}}{\sqrt{n/V}} \cdot VM)$
Lower-Bound	$O((\frac{4^n}{\sqrt{n}})^{(M-1)})$	$O(\frac{4^n}{\sqrt{n}})$	$O(\frac{4^{n/V}}{\sqrt{n/V}})$

	PInv	DefUse	LR-Inv	LR-Def
Upper-Bound	$O(n^2 \cdot \frac{M}{V})$	$O(n^2 \cdot \frac{M}{V} + n)$	$O(n \cdot M)$	$O(n \cdot M)$
Lower-Bound	$O(n \cdot \frac{M}{V})$	$O(n \cdot \frac{M-1}{V})$	$O(1)$	$O(1)$

Table 1: Cost of coverage criteria. (V is the number of variables; M is the number of threads. For simplicity to conduct magnitude-level comparison, we approximate that every thread has n accesses, and has $n_v = \frac{n}{V}$ accesses to each variable.)

6. CONCLUSIONS AND FUTURE WORK

This paper has presented a hierarchy of seven interleaving coverage criteria for concurrent programs and their cost analysis. Our next step is to use real world concurrent programs and concurrency bugs to evaluate the bug exposing capabilities of the proposed coverage criteria, especially those with low cost (quadratic or linear). We will also extend our criteria hierarchy and study how to combine it with the traditional input-selecting coverage criteria.

7. REFERENCES

[1] B. Beizer. *Software Testing Techniques, 2nd edition.* New York: Van Nostrand Reinhold, 1990.

[2] A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur. Applications of synchronization coverage. In *PPoPP*, 2005.

[3] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Trans. Software Eng.*, 15(11):1318–1332, 1989.

[4] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multi-threaded java program test generation. *IBM Systems Journal*, 2002.

[5] M. Factor, E. Farchi, Y. Lichtenstein, and Y. Malka. Testing concurrent programs: a formal evaluation of coverage criteria. In *ICCSSE*, 1996.

[6] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Trans. Softw. Eng.*, 1988.

[7] M. J. Harrold and B. A. Malloy. Data flow testing of parallelized code. In *Proceedings of the International Conference on Software Maintenance*, 1992.

[8] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: Detecting atomicity violations via access interleaving invariants. In *ASPLOS*, 2006.

[9] S. Qadeer and D. Wu. Kiss: keep it simple and sequential. In *PLDI*, 2004.

[10] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM TOCS*, 1997.

[11] R. N. Taylor, D. L. Levine, and C. D. Kelly. Structural testing of concurrent programs. *IEEE Trans. Softw. Eng.*, 1992.

[12] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL*, 2006.

[13] S. N. Weiss. A formal framework for the study of concurrent program testing. In *Proceedings of the Second Workshop on Software Testing, Verification and Analysis*, 1988.

[14] E. J. Weyuker. More experience with data flow testing. *IEEE Trans. Softw. Eng.*, 19(9):912–919, 1993.

[15] C.-S. D. Yang, A. L. Souter, and L. L. Pollock. All-du-path coverage for parallel programs. In *ISSTA*, 1998.

[16] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 1997.