Testing Concurrent Programs: A Formal Evaluation Of Coverage Criteria

Michael Factor
factor@vnet.ibm.com

Eitan Farchi
farchi@vnet.ibm.com

Yossi Lichtenstein,
laor@post.tau.ac.il

Yossi Malka
yossi@vnet.ibm.com

IBM Israel Science and Technology,
Haifa Research Laboratory
MATAM, Haifa 31905, Israel

Abstract

While the literature on testing sequential programs is rich, there has been very little work on testing concurrent programs. One major trend in testing sequential code is to define program-based, test data adequacy criteria. Different criteria are often evaluated using a set of properties which are intended to capture the intuition of what constitutes good testing.

This work's contribution is the adaptation of techniques for evaluating sequential program-based coverage criteria to an abstract concurrent language. We define two sequential and two concurrent program-based criteria which are evaluated with respect to a set of properties. Based upon our analysis, we are able to define new criteria which combine the sequential and concurrent aspects of the program. We also propose a new property that has no parallel in the field of sequential testing. Finally, we discuss the application of the new criteria to the testing of concurrent programs.

1 Introduction

Test data adequacy criteria are an objective means of determining when a program has been sufficiently tested. Examples of such criteria include: "exercise all branches of a program in both directions", "test the extreme points of all input domains"([5]), etc. Most research to date has focused on program-based criteria (i.e., criteria that refer to a program's internal structure) for sequential programs.

Previous work by Weyuker ([10, 11]) established a set of properties that reasonable program-based criteria should satisfy. Parrish and Zweben ([6]) investigated the dependencies between these properties and defined a mathematically strong set of properties. Additional research explored experimental methods to evaluate program-based criteria ([2]). This work has all been in the context of sequential programs, with little discussion of concurrent programs ([9]). However, there is an inherent difference between sequential and concurrent programs: a sequential program's output is independent of speed, whereas a concurrent program's output can be highly dependent on the relative execution order of the component processes.

This work examines how to extend sequential test data criteria to account for this distinction. We adapt Weyuker's technique for evaluating sequential program-based coverage criteria ([10, 11]) to a simple concurrent language. We define two sequential and two concurrent program-based criteria which we evaluate using a consistent set of evaluation properties ([6]). The analysis shows that some properties are not met; satisfying all properties requires criteria that capture both sequential and parallel aspects. We therefore define new criteria that fulfill all properties. We also propose a new property that captures an inherent attribute of concurrent programs.

Taylor, et al. ([9]) describe structural testing of concurrent programs; this differs from our work in two ways. Taylor's criteria are defined with respect to the program's execution state space and not to the program's structure, and Taylor, et al. do not evaluate the quality of their criteria using external properties. However, because they work with the non-computable execution state space of the program and not the tractable space of the program's syntactic structure, Taylor, et al. are able to use a model of concurrent computation that is more realistic (albeit one that is not in general computable).

The literature about formal description techniques for communication protocols contains a rich discussion of testing concurrent code ([3, 1, 7, 8, 12]). However, it mostly dis-

0-8186-7536-5/96 $5.00 © 1996 IEEE
Proceedings of ICCSSE '96

cusses exhaustive conformance testing given a formal specification. Our work aims at concurrent programs where a formal specification is not available and exhaustive testing is not possible.

The next section describes a simple concurrent programming language which serves as the basis for our discussion. Section 3 defines four coverage criteria for this programming language. Section 4 cites the properties used to evaluate sequential coverage criteria. Section 5 is the target of this paper: it uses the properties to evaluate the criteria presented in section 3; the analysis results in defining new and better criteria.

2 A Concurrent Language

This section presents the concurrent programs we wish to discuss. The abstract language, which we describe informally, is not intended to fully define some version of concurrent computing or to replace any existing language. Rather, it aims at emphasizing the difference between testing sequential and concurrent aspects of programs. We model a program by a set of graphs representing the sequential control flow and a partial order representing possible concurrent executions. We also give sequential and parallel composition operators and a partial operational semantics.

A *program* consists of n sequential processes $P_1, \ldots, P_n$ which communicate via channels which are unbounded FIFO queues (following [4]). A process changes its state by one of the following kinds of transitions (i.e., types of instructions): 1) $send(c, m)$ send message m on channel c; 2) $receive(c, m)$ receive message m on channel c; or 3) τ an internal (or hidden) transition, e.g., an assignment to local variables, a conditional forward branch, etc.. We treat the semantics of τ as a black box. If τ is a branch, we assume that both branches can be taken; thus, the set of executions we consider is a superset of the set of executions possible if the semantics of τ are considered.

Message passing is the only means of inter-process communication and synchronization: a $receive(c, m)$ transition may occur only after a $send(c, m)$ (on the same channel) has taken place; otherwise, the $receive$ transition blocks. A $send$ is non-blocking.

We represent a *process*, P, by a rooted, directed acyclic graph (DAG) $\mathcal{G}_P = (V, E)$. We use σ_j^P to refer to the j'th state transition (i.e., instruction) in (sequential) process P. The nodes, V, are the state transitions σ_j^P. The edges, E, represent the flow of control; a transition may have multiple out-going edges if it is a conditional branch. The starting node, σ_0^P, has no incoming edges.

This model is a simplification as it does not allow loops. This implies that each state transition occurs at most once in the execution of a program, and thus, there is no need to distinguish between an instruction and the execution of an instruction. Note that we can capture loops with a fixed bound on the number of iterations, e.g., simple for-loops, by "loop unrolling." The reason behind this limitation, as we discuss below, is to enable us to determine whether or not an input is in the domain of a program. If we were to consider a model that allowed arbitrary loops in the program, we would need to use a weaker definition of a program's input domain that might not allow us to determine whether a given input is in the domain of the program. We should point out that while this limitation may seem extreme, a large class of concurrent code with strenuous testing requirements, e.g., sections of operating system kernels, protocol drivers, etc., contains few if any loops with an arbitrary number of iterations. Another simplification of this model is that it ignores all semantics except $send$ and $receive$ transitions. However in spite of these simplifications, the model is rich enough for our discussion as it allows us to consider the differences between concurrent and sequential programs.

An *execution* of a process P is a path in the graph $\mathcal{G}_P$, starting at σ_0^P and finishing at a leaf; since internal transitions are black boxes, this definition assumes both branches of a condition are possible. An execution of a program `P`, is a set of executions of all its processes. A *trace* of an execution is a sequence (an interleaving) of the transitions taken in the execution. A trace of a program is *consistent* if every $receive(c, m)$ transition has a prior corresponding $send(c, m)$ transition in the trace. A *test* for a program `P` is a consistent trace of `P`.

In the "natural" partial order, for any two concurrent events σ_j^P and σ_k^Q, where σ_j^P is an instance of a $send(c, m1)$ state transition, and σ_k^Q is an instance of a $receive(c, m2)$ state transition, σ_j^P precedes σ_k^Q in the "natural" partial order *iff* for all tests of the program `P`, if σ_j^P and σ_k^Q occur in the trace, than σ_j^P occurs prior to σ_k^Q. Note that we do not include in the partial order relations that are derived from the sequential control flow graph, and thus, internal transitions and those derived from transitivity are not included in the partial order.

Unfortunately, in general, calculating this "natural" partial order from the program text quickly reduces to the halting problem. Instead, we define a program-based partial order that is computable from the program text. A *program-based partial order* $\rightarrow$, exists between two state transitions σ_j^P and σ_k^Q, if σ_j^P is $send(c, m1)$ and σ_k^Q is $receive(c, m2)$ (i.e., any send/receive pair on the same channel). Here too, we do not include relations derived from the sequential control graph. We denote the program-based partial order induced by a

program P by $\rightarrow_P$. The send-receive pairs in the program-based partial order of a program P are a superset of those which are allowed in a test for P.

Since each pair in the natural partial order also appears in our program-based relation, we capture the kind of relationships we would have liked to define using the natural partial order. Our use of a program-based partial order distinguishes our work from Taylor's ([9]). Below, we refer to the program-based partial order as "the partial order". Appendix A.1 gives an example of this partial order, relating it to a trace of a program.

The main drawback of our program-based partial order is that it includes spurious relations which may not only over-constrain but which may not even be feasible. In this respect the natural partial order is obviously preferable. In spite of this, we use our program-based partial order as it is computable. The rest of this work does not depend upon how the partial order is constructed, and if the "natural" partial order was somehow available, its use can only improve the quality of the tests. This also holds for any partial order that is a subset of our program-based partial order and a superset of the "natural" partial order.

Traces are our program's input and output domains; there are no other inputs or outputs. Thus, unlike [6] which uses a single input domain for all programs (a domain which contains all constants representable in the language), in this model the input domain is relative to *a* program. Given the simple language, it is easy to determine if a (finite) trace is included in a program's domain. Obviously if the semantics of internal transitions are taken into account, a trace in the program's domain need not be feasible. In a more sophisticated language, e.g., one allowing arbitrary loops, it may be necessary for all programs to use a single domain of traces, which will include as a subset the feasible traces relative to a given program.

The *parallel composition* of programs $\mathtt{P} = P_1, \ldots, P_n$ and $\mathtt{Q} = Q_1, \ldots, Q_m$ is the program $\mathtt{R} = \mathtt{P} \,||\, \mathtt{Q} = P_1, \ldots, P_n, Q_1, \ldots, Q_m$. The partial order $\rightarrow_R$ for R includes 1) $\rightarrow_P$ and $\rightarrow_Q$ and 2) The new send/receive pairs created by composing P and Q. We can compose a program P with a program E that serves as its *environment* forming a system P||E with no blocked receive transitions. The *pairwise sequential composition* between programs $\mathtt{P} = P_1, \ldots, P_n$ and $\mathtt{Q} = Q_1, \ldots, Q_n$ is the program $\mathtt{R} = \mathtt{P}\,;\,\mathtt{Q} = R_1, \ldots, R_n$. We obtain R_i by adding an edge from each leaf in P_i to the root of a copy of Q_i. We define the partial order $\rightarrow_R$ as for parallel composition.

If $\mathcal{T}$ is a set of tests for P;Q or P||Q, then $\mathcal{T} \downarrow Q$, the projection of $\mathcal{T}$ onto Q, is the set created from each $t \in \mathcal{T}$, by taking either 1) the trace resulting from removing all of P's transitions from t (if the trace is consistent with $\rightarrow_Q$) or 2) an empty test (if by removing P's transitions the trace becomes inconsistent with $\rightarrow_Q$). We define projection identically for sequential and concurrent composition because both add edges to the partial order.

3 Criteria

We define two sequential and two parallel coverage criteria for our programming language. We use the control flow graph to define the sequential criteria and the send/receive partial order to define the parallel criteria.

Consider a program $\mathtt{P} = P_1 \ldots P_n$. Two standard sequential criteria are:

1. **Statement Coverage** (SC): P is statement-covered by a set of tests $\mathcal{T}$, iff for each transition $\sigma_y^{P_k}$ in P, there is a test t in $\mathcal{T}$ such that $\sigma_y^{P_k}$ appears in t.
2. **Path Coverage** (PC): P is path-covered by a set of tests $\mathcal{T}$, iff for each process P_i in P, each path in $\mathcal{G}_{P_i}$, there is a test t in $\mathcal{T}$ such that the execution of t follows the path.

We define two new parallel criteria:

1. **Order Coverage** (OC): P is order-covered by a set of tests $\mathcal{T}$, iff for each pair $\sigma_y^{P_k} \rightarrow \sigma_x^{P_j}$ (such that $k \neq j$) in $\rightarrow_P$, there is a test t in $\mathcal{T}$ such that $\sigma_y^{P_k}$ appears before $\sigma_x^{P_j}$ in t.
2. **Complementary Order Coverage** (COC): P is complementary-order-covered by a set of tests $\mathcal{T}$, iff for each pair $\sigma_y^{P_k}, \sigma_x^{P_j}$ (such that $k \neq j$) which appear in $\rightarrow_P$ and are not comparable under $\rightarrow_P$, there are tests t_1 and t_2 in $\mathcal{T}$ such that $\sigma_x^{P_k}$ appears before $\sigma_y^{P_j}$ in t_1 and $\sigma_x^{P_j}$ appears before $\sigma_y^{P_k}$ in t_2.

Order coverage is analogous to statement coverage. The rationale behind testing all *send-receive* pairs as defined in the partial order $\rightarrow_P$ is the same as the rationale for testing all statements. If no test exercises a particular *send-receive* pair, there is no way for determining if there is a fault in the pair. Because the program-based partial order only approximates the natural partial order, it may contain *send-receive* pairs that are not feasible and thus cannot be covered. There exists a hierarchy of related criteria in which we require chains of three, four, etc., transitions in the partial order to appear in a test. The reason we only consider pairs is the intuition that as one moves up the hierarchy the marginal benefit of the criteria decreases as the cost increases.

Complementary order coverage is more subtle. If two transitions are unrelated by the partial order, the actual order in which they execute should not effect the correctness of an execution. By requiring that a test set exercises both relative execution orders of unrelated transitions, we can increase confidence that there is no hidden and faulty dependency. As with OC, there exists an obvious hierarchy of criteria related to COC in which triples, quadruples, etc., of incomparable transitions are tested in all possible combinations.

The transitions included in $\rightarrow_P$ and used by the OC and COC criteria are *send* and *receive* transitions; the partial order does not include internal transitions. In contrast, the transitions discussed by the SC and PC include both internal and *send* and *receive* transitions. In this sense, the parallel criteria ignore sequential aspects of the program while the sequential criteria cover some parallel aspects. Furthermore, the fact that only *send* and *receive* transitions are explicitly used by the OC and COC criteria makes these criteria less demanding. The partial order taken in concert with the control graph constrains the internal transitions that can occur; however, since there may be multiple paths in the DAG between any two transitions, it does not specify the path.

4 Test Data Adequacy properties

We summarize eight of the twelve test data adequacy properties using the specification independent phrasing of Parrish and Zweben ([6]). We present the properties in their original, *sequential*, terminology. We do not use either a full or an independent set of properties, as our programming language does not contain all constructs needed for some of the properties. In addition, we follow Parrish and Zweben and assume that the empty set is not a program.

The properties refer to a given coverage criterion $\mathcal{C}$. The letter $\mathcal{T}$ denotes a set of tests; P and Q stand for programs. For a program P and a criterion $\mathcal{C}$, $\mathcal{C}(P)$ is the class of all sets of tests that are adequate for $\mathcal{C}$ and P; i.e, if $\mathcal{T} \in \mathcal{C}(\mathcal{P})$, then $\mathcal{T}$ covers P with respect to $\mathcal{C}$. All possible inputs are denoted by $rep(L)$ which denotes all the constants representable in the programming language L. P and Q are renamings if they are semantically and syntactically identical, except perhaps for different variable names. We denote this relation by $Renaming(P, Q)$. If a test set $\mathcal{T}$ causes all *reachable* statements of a program P to be executed, we say that $\mathcal{T}$ *statement covers* P.

We state the properties in first order logic, using the relations mentioned above. We then map these formal (and a-priori meaningless) relations to the programming model, interpreting them in the domain of concurrent testing. For example, above we define $rep(L)$ as a unary relation that denotes all possible inputs to a program; in the next chapter we elaborate on this definition in the context of our concurrent program model.

Property 1 (Applicability) *For every program, there exists an adequate test set.*

$$(\mathbf{APP})\ \forall P \exists \mathcal{T} : \mathcal{T} \in \mathcal{C}(P) \qquad (1)$$

A criterion should be applicable to any program. A criterion that is not applicable for a certain program is not necessarily useless as it may point to a problem with the program itself. For example, consider statement coverage. If a given program P cannot be covered by any test set, some statements in P are not reachable; this may be caused by a programming error.

Property 2 (Non-exhaustive Applicability) *There is a program P and a test set $\mathcal{T}$ such that P is adequately tested by $\mathcal{T}$, and $\mathcal{T}$ is a non-exhaustive test set.*

$$(\mathbf{NA})\ \exists P, \mathcal{T} : \mathcal{T} \in \mathcal{C}(P) \wedge \mathcal{T} \subset [rep(L)] \qquad (2)$$

Demanding that $\mathcal{T}$ is strictly included in $[rep(L)]$ expresses the non-exhaustiveness of $\mathcal{T}$. This property is quite weak: there is at least one program that non-exhaustively meets the criterion.

Property 3 (Monotonicity) *For all programs P and test sets $\mathcal{T}$, $\mathcal{T}'$, if $\mathcal{T}$ is adequate for P and $\mathcal{T}' \supseteq \mathcal{T}$ then $\mathcal{T}'$ is adequate for P.*

$$(\mathbf{MON})\ \forall P, \mathcal{T}, \mathcal{T}' : (\mathcal{T} \in \mathcal{C}(P) \wedge \mathcal{T}' \supseteq \mathcal{T}) \Rightarrow \mathcal{T}' \in \mathcal{C}(P) \qquad (3)$$

Property 4 (Inadequate Empty Set) *The empty set is not an adequate test set for any program.*

$$(\mathbf{IES})\ \forall P : \emptyset \notin \mathcal{C}(P) \qquad (4)$$

Property 5 (Anticomposition) *There exist programs P and Q and a test set $\mathcal{T}$, such that $\mathcal{T}$ is adequate for P and also $\mathcal{T}$ projected to the inputs of Q is adequate for Q, but $\mathcal{T}$ is not adequate for the composition $P; Q$.*[1]

$$(\mathbf{AC})\ \exists P, Q, \mathcal{T}, \mathcal{T}' : (\mathcal{T}' = (\mathcal{T} \downarrow Q)) \wedge \mathcal{T} \in \mathcal{C}(P) \wedge \mathcal{T}' \in \mathcal{C}(Q) \wedge \mathcal{T} \notin \mathcal{C}(P; Q) \qquad (5)$$

[1] Since we don't use a single input domain for all programs, a precise definition of anticomposition should project the test $\mathcal{T}$ to the two components P and Q: $\exists P, Q, \mathcal{T}, \mathcal{T}', \mathcal{T}'' :$ $(\mathcal{T}' = (\mathcal{T} \downarrow Q)) \wedge (\mathcal{T}'' = (\mathcal{T} \downarrow P)) \wedge \mathcal{T}'' \in \mathcal{C}(P) \wedge \mathcal{T}' \in \mathcal{C}(Q) \wedge \mathcal{T} \notin \mathcal{C}(P; Q)$

This property reflects the fact that testing units separately cannot replace the verification of the whole system.

Property 6 (Renaming) *For all programs P and Q such that P is a renaming of Q and for all test sets T, T is adequate for P iff T is adequate for Q.*

$$\textbf{(REN)}\ \forall P, Q, \mathcal{T}: \\ Renamings(P, Q) \Rightarrow (\mathcal{T} \in \mathcal{C}(P) \Leftrightarrow \mathcal{T} \in \mathcal{C}(Q)) \quad (6)$$

Property 7 (Complexity) *For every natural number n, there is a program P such that P is adequately tested by a size n test set, but not by any size $n-1$ test set.*

$$\textbf{(COMP)}\ \forall(n : |rep(L)| \geq n \geq 0)\exists P, \mathcal{T}: \\ |\mathcal{T}| = n \wedge \mathcal{T} \in \mathcal{C}(P) \wedge \\ (\forall \mathcal{T}'(0 \leq |\mathcal{T}'| < n) \Rightarrow \mathcal{T}' \notin \mathcal{C}(P)) \quad (7)$$

Property 8 (Statement Coverage) *For all programs P and test sets $\mathcal{T}$, if $\mathcal{T}$ is adequate for P, then $\mathcal{T}$ causes every executable statement of P to be executed.*

$$\textbf{(SC)}\ \forall P, \mathcal{T} : \mathcal{T} \in \mathcal{C}(P) \Rightarrow \mathcal{T}\ statement\ covers\ P \quad (8)$$

As described by Weyuker ([11]) this property does not imply the statement coverage test adequacy criteria; this property refers only to *executable* statements whereas the criteria refers to all statements. The rationale behind this property is that it requires that a criteria include tests that are related to the program.

5 Analysis

We analyze the four criteria described in section 3 with respect to the properties given in section 4. Table 1 summarizes the results, using a different column for each criteria; we give the Anticomposition (AC) property for both parallel and pairwise sequential compositions. Below we describe and discuss some of the results. Finally, we build new coverage criteria that combine the original such that all properties are satisfied.

Some of the definitions of section 4 need interpretation with respect to our programming language. We define inputs as traces and renaming with respect to channel and message names:

- $rep(L)$ denotes all the inputs representable in the programming language L with respect to the program P being tested; in our case these are all possible *traces* for the given program.[2]

[2]To be precise, we should have used $rep_{\mathtt{P}}(L)$ instead of $rep(L)$ in section 4.

Table 1. Summary of the analysis

	SC	*PC*	*OC*	*COC*
APP	−	−	−	−
NA	+	+	+	+
MON	+	+	+	+
IES	+	+	−	−
AC ;	−	+	+	+
AC ‖	−	−	+	+
REN	+	+	+	+
COMP	+	+	+	+
SC	+	+	−	−

- $Renaming(\mathtt{P}, \mathtt{Q})$ holds when the two programs are semantically and syntactically identical, except perhaps for different variable names. We also allow *channel* and *message* names to be changed by renamings.

5.1 Examples

We consider three examples of detailed analysis of the application of the properties to the criteria as summarized in table 1.

Applicability for Complementary Order Coverage: Consider a program with two processes, $\mathtt{P} = P_1, P_2$. Let each process include a single (reachable) *receive* transition: $P_1 = receive(c, n)$ and $P_2 = receive(d, m)$. As both transitions are *receives*, the set of consistent traces for P is empty; thus any set of tests for P is empty. On the other hand, both transitions $receive(c, n)$ and $receive(d, m)$ are incomparable in $\rightarrow_{\mathtt{P}}$ thus contradicting the applicability property for the Complementary Order Coverage.

Inadequate Empty Set for Order Coverage: Consider $\mathtt{P} = P_1, P_2$; each process containing only one transition: $P_1 = (a1)$ and $P_2 = (a2)$. Let $a1$ and $a2$ be internal transitions; the set of comparable events under $\rightarrow_{\mathtt{P}}$ is empty. Thus, the empty test set covers $OC(\mathtt{P})$ contradicting IES.

Anticomposition (of parallel composition) for Path Coverage: Consider the composition of any two programs P||Q. 1) The parallel composition does not change the paths traversed separately in each of the programs. Thus, each path in P||Q is either a path in P or a path in Q. 2) If T is a set of tests for P||Q with $T \downarrow \mathtt{P}$ (i.e., projection on P) that path-covers P, then T includes a subset of tests that covers the paths in P||Q that originated from P. The same argument is true for Q. 3) Thus, for any test set T for P||Q with $T \downarrow \mathtt{P}$

that path covers P (and $T \downarrow$ Q that covers Q), (by 2) T includes a subset of tests that covers the set of paths in P||Q that originated in either P or Q. Thus, (by 1) T includes a path coverage of P||Q and **AC** is not met.[3]

5.2 Discussion

Applicability: The criteria definitions are coarse and do not treat unreachable transitions correctly. It is easy to refine the statement and path criteria to exclude inapplicable statements (as done in [13]). However, it is less straightforward to exclude unreachable transitions excluded from the criteria phrased in the partial order terms. In this sense, the inapplicability of the *Order* and *Complementary* criteria are harder to mend.

Inadequate Empty Set: As the *Order* criterion is defined on the partial order, in the case of an empty order, an empty test set is adequate. In the same way, when the order is full (and the complementary order is empty) an empty test is adequate with respect to the *Complementary* criterion. The parallel criteria fails the **IES** property because a program may be non-empty from the sequential point of view and empty with respect to inter-process communication.

Statement Coverage : This property serves as a minimum requirement for any type of sequential coverage. The parallel criteria do not comply with this minimum as they cover a different domain. Thus, instead of looking at where the parallel criteria cover *all* statements, the concurrent criteria could consider only statements related to interactions between processes, i.e., those occurring in the partial order.

Anticomposition : There is an intuitive difference between composition of sequential and parallel code. For sequential code, it is sufficient to provide the inputs to execute a unit; by contrast, a unit of parallel code may also require an environment, a set of messages, which the unit receives and for which it does not contain corresponding *send*s. Without this environment, which can be provided by the component's second unit, there may be no tests for the unit, i.e., no consistent traces. Thus, both OC and COC trivially fulfill anti-composition — there are programs in which the program as a whole can be adequately tested but in which the individual units cannot be adequately tested as they have no environment. This differs from sequential code, where anti-composition is used to distinguish between a weak criteria (statement coverage) and a strong criteria (path coverage).

[3] However, detailed semantics of the programming language may change this result. If forward branches depend on the content of received messages, a path traversed in P (by itself) will not necessarily be the same when traversed for a trace of P||Q.

5.3 New Criteria

The above analysis can lead us to criteria that satisfy all of the discussed properties. First, we change the domain of the tests to guarantee that all of the transitions are reachable. We redefine the domain of program P's tests to be all of the tests of a program P||E, for any environment E.

It suffices to use an environment consisting of a separate process executing $send(c, m)$ for each $receive(c, m)$ in program P. This guarantees all $receive$s can complete, making it possible for the execution to continue, and leading all transitions to be reachable. As stated earlier, since we assume a lack of knowledge about the semantics of the control flow graph, barring blocking $receive$s, all transitions are inherently reachable. To further motivate this, since the properties always use the $\exists$ qualifier on the set of tests T, our redefinition of the domain of the program's tests includes the sufficient environment consisting of one $send$ (in its own process) per $receive$ in the original program. Thus, if instead of changing the domain to be for any environment E, we had changed the domain to use the environment consisting of a single $send$ (in its own process) per $receive$ in the original program, the outcome of the analysis would not change.

Note that this definition of an environment is more general than the environment we would like to use, however, it is an environment that we can statically calculate from the program's text. The problem is that a $receive$ in the program may be satisfied by a $send$ from the environment, whereas it, in some sense, should be satisfied by a $send$ from another process in the program. The ideal environment would be one in which only those $send$s which are required to come from an external source are included in the environment.[4]

With this revised definition of the domain of the tests, the criteria SC satisfies the applicability (**APP**) property; it already satisfied the statement coverage (**SC**) and inadequate empty set (**IES**) properties. Order Coverage (as well as Complementary Order Coverage) already satisfy the anti-composition property (**AC**). Thus, combining statement coverage and one of the order criteria, in the context of the revised domain of tests, provides the framework for satisfying all of the criteria. Table 2 summarizes the results of this analysis; an additional, necessary simple definition is given:

Coverage combination: A program P is $(A \wedge B)$-covered by a set of tests T, iff P is A-covered by T **and** P is B-covered by T.

Note that when a property is satisfied separately by a crite-

[4] To use such an "ideal" environment, we would also need to use a more restrictive partial order than the program-based partial order, one which did not include spurious relations.

Table 2. New Criteria

	SC	*OC*	*COC*	*SC*∧ *OC*	*SC*∧ *COC*
APP	+	+	+	+	+
NA	+	+	+	+	+
MON	+	+	+	+	+
IES	+	−	−	+	+
AC ;	−	+	+	+	+
AC ‖	−	+	+	+	+
REN	+	+	+	+	+
COMP	+	+	+	+	+
SC	+	−	−	+	+

rion A and by a criterion B, it does not necessarily follow that $A \wedge B$ satisfies the property. For example, some of the properties are stated by an exists quantification ($\exists$P). Generally, the program used for A in such a case is different from the program for B, and there might be no program satisfying the property for $A \wedge B$.

5.4 New Property

We should also consider properties which may be added to the current set. In particular, are there additional properties which are important for testing concurrent programs and are irrelevant for sequential programs?

We identify one such property:- *Repeatability*. If a test set covers a program with respect to a certain criterion, this should be repeated in subsequent applications of the same test set to the program. This is trivial for deterministic programs, but it is a problem for non-deterministic programs. All criteria fail this property if the tests do not resolve the non-deterministic decisions. Since we defined tests as traces, i.e., interleavings of executions, resolving possible non-determinism, we have made our criteria repeatable.

6 Conclusion

The main contribution of this work is the systematic evaluation of program-based coverage criteria in the context of concurrent programs. We have adopted Weyuker's formal approach to the domain of testing concurrent programs.

The systematic evaluation of coverage criteria is useful; trying to satisfy all properties led us to the definition of new and better criteria. In particular, it has become apparent that a combination of sequential and concurrent criteria is required to satisfy all properties. The evaluation process clarified the need to combine the two aspects into a single criterion.

The order and complementary order criteria make explicit the difference between testing concurrent and sequential programs. The partial order that describes the interactions between processes has a complementary graph that is important for testing concurrent programs. It expresses possible interleavings that must be tested. For sequential programs, the complement of the control flow graph has no such meaning.

Future work should look into using a partial order that is more restrictive than the program-based partial order, i.e., one that contains fewer spurious relations, but yet is still computable from the syntax of the program. One possible approach might be to eliminate those relations that contradict the serial control flow graph, perhaps taking into account transitivity. A related item of future work, that is dependent upon using a more restrictive partial order, would be to look at a more restrictive environment as described in section 5.3.

In addition further research is needed to apply this approach to a more complete model of concurrent programs. As described in section 2 a more general model, e.g., one that allowed arbitrary loops, would require a weaker definition of the domain of the program. On the surface, to modify the model to allow arbitrary loops, all that is required is to redefine a process to be a rooted, directed graph, instead of a rooted, DAG; work is still required to verify that this generalization does not impact the definition of the partial order. However, it is not obvious how such a generalized model could allow a computable determination of whether an input is in a program's domain; further work is required to see how much we can extend the model while still allowing this determination to be computable. In addition, if we extend the model as described, the criteria must be reevaluated with respect to the properties to see that they continue to hold; given a generalized model, it will be interesting to determine if new properties are required to differentiate parallel from sequential criteria. Finally, given a more complete model, it will be possible to address the question of the complexity of using the criteria.

References

[1] A. Aho. An optimization technique for protocol conformance test generation based on UIO sequences and rural chinese postman tours. In *Protocol Specification, Testing, and Verification*, 1989.

[2] P. Frankl and S. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transaction on Software Engineering*, 19(8), August 1993.

[3] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[4] ISO. Lotos - a formal description technique based on temporal ordering of observed behavior. Technical report, International Standard Organization, 1988.

[5] B. Jeng and E. Weyuker. A simplified domain-testing strategy. *ACM Transaction on Software Engineering and Methodology*, 3(3), July 1994.

[6] A. Parrish and S. Zweben. Analysis and refinement of software test data adequacy properties. *IEEE Transaction on Software Engineering*, 17(6), June 1991.

[7] D. Sidhu. Formal methods for protocol testing: A detailed study. *IEEE Transactions on Software Engineering*, 15(4), 1989.

[8] D. Sidhu. Protocol testing, the first ten years, the next ten years. In *Protocol Specification, Testing, and Verification*, 1990.

[9] R. Taylor, D. Levine, and C. Kelly. Structural testing of concurrent programs. *IEEE Transactions on Software Engineering*, 18(3), March 1992.

[10] E. Weyuker. Axiomatizing software test data adequacy. *IEEE Transaction on Software Engineering*, SE-12(12), December 1986.

[11] E. Weyuker. The evaluation of program-based software test data adequacy criteria. *Communications of the ACM*, 31(6), June 1988.

[12] C. Wezeman. The co-op method for compositional derivation of conformance testers. In *Protocol Specification, Testing, and Verification*, 1990.

[13] S. Zweben and J. Gourlay. On the adequacy of Weyuker's test data adequacy axioms. *IEEE Transaction on Software Engineering*, 15(4), April 1989.

A Appendix

A.1 Program Based Partial Order; An Example

As an example for the program based partial order, consider the following program `P`:

```
        P1                  P2

send(c,m1)          receive(c,m1)
send(c,m2)          receive(c,m2)
```

It contains two processes using one channel. The partial order contains all send/receive pairs:

```
send(c,m1)->receive(c,m1)
send(c,m1)->receive(c,m2)
send(c,m2)->receive(c,m1)
send(c,m2)->receive(c,m2)
```

There are 4!=24 possible combinations of the four transitions. Eighteen of them are removed by the control flow DAG (e.g., `send(c,m2) send(c,m1)`) because they are not interleavings of executions of the processes. Out of the remaining six interleaving only two are consistent traces:

```
t1:  send(c,m1) receive(c,m1)
        send(c,m2) receive(c,m2)

t2:  send(c,m1) send(c,m2)
        receive(c,m1) receive(c,m2)
```

Note that part of the `t1` trace seems to be inconsistent with the partial order $\rightarrow_P$ (i.e., `receive(c,m1) send(c,m2)`); however, it is actually a consistent trace as every receive does have a prior corresponding send transition. The partial order is defined to represent only dependencies that stem from the communication on each of the channels. It does not represent the order imposed by the control flow graph. Since the partial order does not represent the dependencies that exist over all executions, it may not be extensible to the total order defined by a consistent trace.