Better Predicate Testing

Gary Kaminski
Software Engineering
George Mason University
Fairfax VA, USA
gkaminsk@gmu.edu

Paul Ammann
Software Engineering
George Mason University
Fairfax VA, USA
pammann@gmu.edu

Jeff Offutt
Software Engineering
George Mason University
Fairfax VA, USA
offutt@gmu.edu

ABSTRACT
Mutation testing is widely recognized as being extremely powerful, but is considered difficult to automate enough for practical use. This paper theoretically addresses two possible reasons for this: the generation of redundant mutants and the lack of integration of mutation analysis with other test criteria. By addressing these two issues, this paper brings an important mutation operator, relational-operator-replacement (ROR), closer to practical use. First, we develop fault hierarchies for the six relational operators, each of which generates seven mutants per clause. These hierarchies show that, for any given clause, only three mutants are necessary. This theoretical result can be integrated easily into mutation analysis tools, thereby eliminating generation of 57% of the ROR mutants. Second, we show how to bring the power of the ROR operator to the widely used Multiple Condition-Decision Coverage (MCDC) test criterion. This theoretical result includes an algorithm to transform any MCDC-adequate test set into a test set that also satisfies RORG, a new version of ROR appropriate for the MCDC context. The transformation does not use traditional mutation analysis, so can easily be integrated into existing MCDC tools and processes.

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

General Terms
Verification

Keywords
Logic testing, Mutation

1. INTRODUCTION
The ability for mutation testing [7, 8] to help testers design high quality tests has always depended directly on the mutation operators. In program-based mutation testing, a mutation operator is a rule that specifies changes to syntactic elements in a program. A well designed set of operators can result in very powerful testing, but a poorly designed set can result in ineffective tests. Mutation operators have been designed for several programming languages, including COBOL [10], Fortran 77 [8, 19], C [6], Ada [25], and Java [18, 22]. Jia and Harman recently surveyed mutation analysis for programs and other software engineering artifacts [12]. The statement-level operators have been fairly stable since the Mothra project [19], with the most important suggestion for change being from the selective operator study [23], where it was found that using five Mothra mutation operators would yield tests that killed most other mutants.

However, users of mutation have observed that mutation creates many test requirements (that is, mutants) relative to the number of tests needed when compared to other test criteria. For example, Li et al. found that mutation yielded fewer tests than the edge-pair, all-uses and prime path criteria, even though it had far more test requirements [21]. The only reasonable conclusion from this result is that the mutants somehow “overlap” in the tests needed to kill them, and probably in their ability to find faults. In other words, some mutants appear to be redundant.

The development of fault detection hierarchies, most notably the DNF fault hierarchy of Lau and Yu [20], offers a way to deal with the problem of redundant mutants. A fault hierarchy describes a dominance relation among fault classes. If a test set detects faults in a given class in the hierarchy, those tests are also guaranteed to detect faults in classes that class dominates. From the mutation analysis perspective, this means that if a fault hierarchy is built for a set of mutants, there is no need to generate mutants that are dominated by other mutants.

A closely related approach to increasing the effectiveness of each mutant is the notion of a subsuming Higher-Order-Mutant (HOM) [11]. A subsuming HOM is built by combining several mutations in such a way that the combined mutant (the HOM) subsumes all of its constituent mutants. A key property of a subsuming HOM is that a test that kills the HOM is also guaranteed to kill the subsumed mutants, and hence these mutants do not need to be generated. Kaminski and Ammann put both of these ideas together in the context of mutation analysis of logic expressions in disjunctive normal form (DNF) [13]. They showed that not only are many redundant mutants generated for logic expressions, worse, these mutants miss many faults in the Lau and Yu fault hierarchy. They further showed how to automatically con-
struct a small number of subsuming HOMs that guarantee
detection of the entire Lau and Yu fault hierarchy—and, by
subsumption, all mutants generated by a typical mutation
analysis tool on the boolean structure of logic expressions.

This paper considers the relational operators that com-
monly appear in boolean expressions. In terms of muta-
tion analysis, this means considering the relational operator
replacement (ROR) operator. Our first contribution is to
develop fault hierarchies for the mutants generated by the
ROR operator. These hierarchies show that of the seven mu-
tants generated by a single application of an ROR operator,
only three are necessary.

The second contribution is to increase the strength of the
logic coverage test criterion of Multiple Condition-Decision
Coverage (MCDC) [5]. Logic coverage criteria such as MCDC
and ACC [5, 2] test predicates at the clause level. That is,
in a predicate \( p = a \land b \lor c \), logic criteria test the individ-
ual clauses \((a, b, c)\). Mutation analysis, however, uses the
ROR operator to test at a lower level of abstraction, in-
side the clauses. Thus, if \( a \equiv (x > y), b \equiv (m <= n)\),
and \( c \equiv (d = e)\), ROR tests whether the relations inside
the clauses are formulated correctly. This paper defines a
stronger version of MCDC that leverages the ROR fault hi-
erarchies to integrate the power of the ROR mutation op-
erator into the logic criteria. This result includes an algorithm
that augments an MCDC-adequate test set to also be ade-
quate with respect to RORG, a variant of ROR appropriate
for integration with MCDC.

MCDC, which is equivalent to Active Clause Coverage
(ACC) [2, 1], is required by the US Federal Aviation Ad-
ministration to test safety critical systems [26], and comes in
several slightly varying versions [4]. As it turns out, any
ROR-adequate test set is guaranteed also to satisfy the
weakest version of MCDC [1]. However, the converse is not
true; an MCDC-adequate test set does not necessarily sat-
sify ROR coverage. Intuitively, this is because of the differ-
eence in the previous paragraph; MCDC treats predicates as
boolean functions and ignores relational operators in these
expressions. For applications where faults in relational op-
erators are a concern, including ROR can clearly lead to
stronger tests.

The organization of this paper is as follows. Section 2
describes background in mutation operators, logic mutation
operators, and recent results in logic testing. Section 3 ex-
plores the relationship between the ROR mutation operator
and logic criteria and section 4 presents a new ROR fault
hierarchy. Section 5 presents a new version of the ROR op-
erator that is just as effective but that produces less than
half the number of mutants. Section 6 presents modified
versions of MCDC that are more effective and only slightly
more expensive in terms of tests required. Section 7 offers
conclusions and recommendations for future automation.

2. BACKGROUND

The classic definition of the ROR operator is from the
1991 detailed description of the Mothra mutation system
[19]: Each occurrence of a relational operator \((<, >, \leq, \geq,
\neq, =)\) is replaced by each other operator and the expression
is replaced by True and False. Most mutation systems (in-
cluding muJava [22]) since Mothra have implemented these
operators following these definitions.

The literature contains several test coverage criteria that
focus on the logical structure of predicates. This literature
assumes that a predicate is assembled from boolean valued
clauses via the standard boolean operators, typically in-
cluding not (\(-\)), and (\(\land\)), and or (\(\lor\)). Significantly, from
the perspective of this paper, the internal structure of the
clauses is ignored. The most powerful logic coverage crite-
ria is combinatorial coverage, which requires every possible
assignment of truth values to clauses. In a predicate with \(n\)
clauses, combinatorial coverage requires \(2^n\) tests.

Other logic coverage criteria ask for some subset of the
possible \(2^n\) tests defined by combinatorial coverage. Clause
coverage requires each clause to take on the values true and
false, and can be satisfied with just two tests. Predicate
coverage requires the predicate as a whole to take on the
values true and false, and can also be satisfied with just
two tests. Clause coverage and predicate coverage are both
fairly weak, and neither subsumes the other.

Modified Condition Decision Coverage (MCDC) [5], which
is equivalent to Active Clause Coverage (ACC) [1], is widely
perceived as a powerful test coverage technique, and is used
in the certification of safety critical systems [26]. The idea
behind MCDC is that each clause should be tested to be
both true and false under circumstances where the clause
“matters,” where this is interpreted to mean that chang-
ing the value of the clause necessarily changes the value of
the predicate. Note that MCDC test requirements come in
pairs—one requirement for the clause true and one for the
clause false. MCDC comes in various forms depending on
whether each test pair faces additional constraints beyond
what is mentioned above, but the details of these forms are
independent of the contributions of this paper.

A different approach to logic coverage bases it on fault
detection power with respect to the Lau and Yu fault hi-
erarchy [20]. The most powerful of these techniques is
MUMCUT [3], which is guaranteed to detect the entire
hierarchy. MUMCUT generally only applies to predicates
written in Disjunctive Normal Form (DNF), although extent-
sions to more general forms have been explored [27]. MUM-
CUT generates many unnecessary tests; various refinements
of MUMCUT have addressed this shortcoming [9, 14, 15, 16].

3. ROR AND LOGIC CRITERIA

This section proves that logic coverage criteria do not sub-
sume ROR mutation testing. For convenience, we use the
term ROR mutation to mean mutation using just the ROR
operator. The strongest logic coverage criterion is combina-
torial coverage, which requires that a predicate be tested
with all combinations of truth values. Consider the two-
clause predicate \(a < b \lor c < d\). Combinatorial coverage
requires the tests TT, TF, FT, FF. The following assign-
ments to \(a, b, c, d\) satisfy combinatorial coverage:

\[
\begin{align*}
a &= 1, b = 2, c = 1, d = 2 & (TT) \\
a &= 1, b = 2, c = 2, d = 1 & (TF) \\
a &= 2, b = 1, c = 1, d = 2 & (FT) \\
a &= 2, b = 1, c = 2, d = 1 & (FF)
\end{align*}
\]

However, none of these four tests detects either of the
two ROR mutants where \(c\) is replaced by \(\leq\). To detect
the mutant where \(a < b\) is changed to \(a <= b\), \(a\) and \(b\)
must have the same value. Thus, combinatorial coverage
does not subsume ROR mutation. Combinatorial coverage
is the strongest logic criterion and subsumes all others, thus
no logic coverage criterion can subsume ROR mutation. In-
tuitively, the logic coverage criteria treat each clause as a
unit, as a boolean variable, ignoring any structure inside the clause such as the relational operators. The ROR operator, on the other hand, explicitly requires tests to evaluate whether the correct relational operator was used. That is, logic criteria test the clause level, whereas the ROR operator tests the relational level, a more detailed level of abstraction.

4. A NEW ROR FAULT HIERARCHY

Mutation is widely considered to be expensive, but “expense” can be measured in several ways. Li et al. [21] found that although mutation creates more test requirements (that is, mutants) than other test criteria, it does not need more tests. The obvious implication from that result is that many mutants are unnecessary or redundant. The selective operator study [23] reduced the number of mutants by an order of magnitude, but mutation systems such as muJava [22] still generate more mutants than are necessary. (Li et al. ’s study used the muJava selective set of mutation operators.)

This section analyzes the ROR mutation operator on a case by case basis. For each relational operator, the conditions under which mutants created by that operator will be killed are derived. These are called detection conditions.

From that, a hierarchy of mutants is formed. The table in figure 1 shows the detection conditions for each mutant of the < operator. When \( a < b \) is mutated to False, the detection condition is \( a < b \) and the value for \( a < b \) must be True. Likewise, if the mutant is \( a <= b \), the detection condition is \( a == b \) and the value must be False.

This leads to the mutant class hierarchy for < in figure 1. The arrows imply a dominance in the sense of Lau and Yu [20]. If a test kills the mutant where < is replaced by False, that test is guaranteed to also kill the mutants where < is replaced by == and >, and by transitivity, the mutant where < is replaced by >=.

<table>
<thead>
<tr>
<th>Mutant</th>
<th>Detection condition</th>
<th>Value of ( a &lt; b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; replaced by F</td>
<td>( a &lt; b )</td>
<td>T</td>
</tr>
<tr>
<td>&lt; replaced by &lt;=</td>
<td>( a == b )</td>
<td>F</td>
</tr>
<tr>
<td>&lt; replaced by !=</td>
<td>( a &gt; b )</td>
<td>F</td>
</tr>
<tr>
<td>&lt; replaced by ==</td>
<td>( a &lt;= b )</td>
<td>T or F</td>
</tr>
<tr>
<td>&lt; replaced by &gt;</td>
<td>( a &gt; b )</td>
<td>T or F</td>
</tr>
<tr>
<td>&lt; replaced by &gt;=</td>
<td>T</td>
<td>T or F</td>
</tr>
</tbody>
</table>

Figure 1: Mutants, Detection Conditions, and Class Hierarchy for <

For all six relational operators, we can immediately see that tests that detect three of the ROR mutants are guaranteed to detect all seven ROR mutants. Conversely, if there is no arrow in a hierarchy from one mutant to another, a test that detects the first mutant is guaranteed not to detect the other. For example, consider figure 1. A test that detects the mutant where < is replaced with <= will never detect the mutant where < is replaced with >. Also any test that detects a mutant at the top level of a hierarchy is guaranteed not to detect either of the other two mutants at the top level of the hierarchy. For example, again consider figure 1. A test that detects the mutant where < is replaced with != will never detect the mutant where < is replaced with False.

Detecting all ROR mutants guarantees clause coverage. Both T and F appear at least once among the top three rows in the “Value of a relop b” column in each table, which achieves clause coverage.
5. A CHEAPER ROR OPERATOR

The detection conditions and mutant hierarchies in section 4 lead to an immediate result. The classic definition of the ROR operator [19] should be reformulated to only create the three mutants on top of the mutant hierarchy for the relational operator being mutated. Specifically, if the operator is <, only <, <=, and False should be created; if the operator is >, only >, >=, and False should be created; if the operator is <=, only <=, ==, and True should be created; if the operator is >=, only >=, >, ==, and True should be created; if the operator is ==, only ==, !=, and False should be created; and if the operator is !=, only <, >, and True should be created. This is an immediate savings of four mutants for each relational operator.

6. STRONGER LOGIC CRITERIA

As described in section 3, logic coverage criteria such as MCDC, ACC, and MUMCUT [3] (an extremely powerful
tants are: \( p = a_1 \geq a_2 \lor (b_1 = b_2 \land a_1 > a_2) \) and \( p = a_1 = a_2 \lor (b_1 = b_2 \land a_1 = a_2) \).

With the RORG adaptation of ROR, we can develop an algorithm that adds RORG-adequacy to an MCDC test set. The idea behind the algorithm is quite simple. We identify tests that satisfy MCDC with respect to each boolean clause \( c \). The MCDC requirements on these tests are that \( c \) take on the values True and False under conditions where \( c \) determines the value of predicate \( p \). We then note that the top three rows of every table in section 4 have the same three detection conditions, \(<, ==, \) and \( > \). MCDC by itself requires two tests for each clause (True and False). So at least two of these will have been satisfied by a test where \( c \) determines the value of \( p \). First, the algorithm identifies whether one of the three detection conditions is not satisfied by the MCDC tests, and which. Then it adds a test to satisfy the third. This process is shown in pseudo-code form in algorithm 1.

**Algorithm 1: Algorithm to Make MCDC Test Set RORG-Adequate**

**Require:** Predicate \( p \) and a test set \( T \) that satisfies MCDC (ACC) with respect to \( p \)

**Ensure:** A test set that still satisfies MCDC (ACC), but is now also RORG-adequate

1: // It does not matter which version of ACC is satisfied
2: // (GACC, CACC, or RACC), or whether masking or non-masking MCDC is used.
3: // non-masking MCDC is used.
4: for each clause \( c \) in \( p \) do
5: if \( c \) contains a relational operator \( \text{relop} \) then
6: Identify \( T_c \), the tests for which \( c \) determines \( p \)
7: // Clause \( c \) determines predicate \( p \) if changing
8: // the value of \( c \), while leaving all other
9: // clauses unchanged, changes the value of \( p \) [1].
10: // \( T_c \) will have at least two tests and possibly more.
11: // \( c \) will have the value True for at least one test
12: // and False for at least one test.
13: Assume \( c = c_1 \text{relop} c_2 \)
14: // We need three tests, \( c_1 < c_2, c_1 == c_2 \), and \( c_1 > c_2 \), and are assured of having at least two.
15: for each test \( t_i \) in \( T_c \) do
16: isCovered[\( \langle \rangle \)]=isCovered[\( == \)]=isCovered[\( > \)]=False
17: for each relop in \( \langle \langle, ==, > \rangle \) do
18: if \( c_1 \text{relop} c_2 \) is True for \( t_i \), then
19: isCovered[\( \text{relop} \)]=True
20: end if (\( c_1 \text{relop} c_2 \) is True)
21: end for (each relop)
22: end for (each test)
23: for each relop in \( \langle \langle, ==, > \rangle \) do
24: if isCovered[\( \text{relop} \)]=False then
25: Construct a new test by modifying an arbitrary test in \( T_c \). Leave all other variables alone, but change the values for the variables in \( c \) so that \( c_1 \text{relop} c_2 \) is True
26: end if (isCovered[\( == \)]=False)
27: end for (each relop)
28: end if (\( c \) contains a relop)
30: end for (each clause)

**Algorithm Proof Sketch:** The algorithm makes two claims about the resulting test set. The first is that it satisfies MCDC. Since the input test set satisfies MCDC, and no tests are removed from this set, it is clear that the output still satisfies MCDC.

The second claim is that the output test set kills every RORG mutant. We prove this via weak mutation analysis, which is arguably the only generally applicable approach in the context of MCDC. Weak mutation analysis requires infection of the subsequent state to kill a mutant. Offutt and Lee [24] found that infection in the context of a mutated predicate is best defined as the final value of the predicate being incorrect. Thus, for each RORG mutant there must be at least one test that causes the predicate to evaluate to the wrong value. Inspection of the detection conditions in figures 1 through 6 shows that to kill all ROR mutants, the relation between \( c_1 \) and \( c_2 \) in \( c = c_1 \text{relop} c_2 \) must have all the three possible values, namely: \( c_1 < c_2, c_1 == c_2, \) and \( c_1 > c_2 \). For these detection conditions to cause predicate \( p \) to evaluate to a different result, it is necessary that \( c \) determines \( p \). The algorithm forces the three possible relations between \( c_1 \) and \( c_2 \) for tests where \( c \) determines \( p \). Hence RORG is satisfied. □

Note that the resulting test set does not necessarily satisfy ROR. The reason is that if we consider each clause \( c \) syntactically as it appears in the predicate, there is no guarantee that the original MCDC test set has any tests in the set \( T_c \). However, for RORG, each \( T_c \) is guaranteed to always be at least of size 2, and, further, to satisfy predicate coverage. Complexity is on the order of the product of the number of tests and the number of clauses. If the test set is specifically optimized for MCDC, the size of the test set is linear in the number of clauses. Hence, in this case, the algorithm is quadratic in the number of clauses.

**Example:** Consider the predicate \( p = a \land b \lor c \), where \( a = (a_1 < a_2), b = (b_1 \leq b_2), \) and \( c = (c_1 == c_2) \).

The following test set satisfies the most restrictive MCDC coverage criterion (RACC): \( T = \{t_1, t_2, t_3, t_4\} = \{TTT, TFT, TFF, TFF\} \). These tests are refined to have the value assignments:

\[
\begin{align*}
t_1: & a_1 = 5, a_2 = 6, b_1 = 10, b_2 = 11, c_1 = 21, c_2 = 22 \\
t_2: & a_1 = 5, a_2 = 6, b_1 = 11, b_2 = 10, c_1 = 21, c_2 = 21 \\
t_3: & a_1 = 5, a_2 = 6, b_1 = 11, b_2 = 10, c_1 = 21, c_2 = 22 \\
t_4: & a_1 = 5, a_2 = 6, b_1 = 11, b_2 = 11, c_1 = 21, c_2 = 22
\end{align*}
\]

We first consider clause \( a \) from line 4 in algorithm 1. The set \( T_a \) of tests where \( a \) determines \( p \) is \( \{t_1, t_4\} \). Test \( t_1 \) satisfies \( a_1 < a_2 \) and \( t_4 \) satisfies \( a_1 > a_2 \), so the algorithm adds a new test (in line 26) to satisfy \( a_1 == a_2 \). Any test for which \( a \) determines \( p \) will do, so the algorithm starts with \( t_1 \), and modifies the values for \( a_1 \) and \( a_2 \) to be \( a_1 = 5 \) and \( a_2 = 5 \).

For clause \( b \), the set \( T_b \) of tests where \( b \) determines \( p \) is \( \{t_1, t_3\} \). Test \( t_1 \) satisfies \( b_1 < b_2 \) and \( t_3 \) satisfies \( b_1 > b_2 \), so the algorithm adds a new test (in line 26) to satisfy \( b_1 == b_2 \). Any test for which \( b \) determines \( p \) will do, so the algorithm starts with \( t_1 \), and modifies the values for \( b_1 \) and \( b_2 \) to be \( b_1 = 10 \) and \( b_2 = 10 \).

Finally, for clause \( c \), the set \( T_c \) of tests where \( c \) determines \( p \) is \( \{t_2, t_3\} \). Test \( t_2 \) satisfies \( c_1 == c_2 \) and \( t_3 \) satisfies \( c_1 > c_2 \), so the algorithm adds a new test (in line 26) to satisfy \( c_1 > c_2 \). Any test for which \( c \) determines \( p \) will do, so the algorithm starts with \( t_2 \), and modifies the values for \( c_1 \) and \( c_2 \) to be \( c_1 = 22 \) and \( c_2 = 21 \).
7. CONCLUSIONS AND RECOMMENDATIONS

This paper presents two theoretical results, a way to reduce the number of mutants generated for the ROR operator by eliminating redundancy among mutants, and a way to strengthen logic criteria such as MCDC by using the ROR mutation operator to increase precision.

The first result can be used to improve future mutation tools by reducing the number of mutants generated. The traditional ROR operator creates seven mutants for each relational operator, of which four are provably redundant. We believe it is likely that existing mutation systems produce lots of similarly redundant mutants, and similar analysis could greatly reduce the number of mutants created by future mutation systems. In mutation testing, each mutant represents a test requirement, so reducing the number of mutants created can have a major impact on the automation of mutation testing.

The second result can be used to strengthen logic test criteria. Logic testing criteria have traditionally only looked at the clause level, treating each clause as a simple boolean variable. Part of the power of mutation stems from the fact that it looks inside clauses, and tries to determine whether a clause such as \( a > b \) is correctly formulated. By adding one more test for each clause, and using the redundancy proofs for the ROR operator, this paper shows how logic criteria can be extended to gain this power. The algorithm in section 6 shows how this technique can easily be incorporated into an automated test tool.

This result is particularly significant for the MCDC criterion [5], because it is mandated for certain safety-critical software components on aircrafts and air traffic controllers by the US Federal Aviation Administration [26]. The simple extension to MCDC presented in this paper has the ability to strengthen testing of this crucial software, potentially making air travel safer. Although the algorithm is definitive in terms of adding ROR-adequacy to an MCDC-adequate test set, empirical studies are needed to determine how much augmenting MCDC test sets with ROR-adequacy improves fault detection. The algorithm can also be used in the automation of testing virtually any kind of control software, much of which makes heavy use of logical predicates and much of which is safety critical.

<table>
<thead>
<tr>
<th>Test Value</th>
<th>( t_1 )</th>
<th>( t_2 )</th>
<th>( t_3 )</th>
<th>( t_4 )</th>
<th>New ((t_1))</th>
<th>New ((t_2))</th>
<th>New ((t_3))</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a \neq b )</td>
<td>TTT</td>
<td>TTT</td>
<td>TTT</td>
<td>TTT</td>
<td>TTT</td>
<td>TTT</td>
<td>TTT</td>
</tr>
<tr>
<td>( a = b )</td>
<td>TTT</td>
<td>TTT</td>
<td>TTT</td>
<td>TTT</td>
<td>TTT</td>
<td>TTT</td>
<td>TTT</td>
</tr>
<tr>
<td>( a &lt; b )</td>
<td>TTT</td>
<td>TTT</td>
<td>TTT</td>
<td>TTT</td>
<td>TTT</td>
<td>TTT</td>
<td>TTT</td>
</tr>
<tr>
<td>( a &gt; b )</td>
<td>TTT</td>
<td>TTT</td>
<td>TTT</td>
<td>TTT</td>
<td>TTT</td>
<td>TTT</td>
<td>TTT</td>
</tr>
</tbody>
</table>

8. REFERENCES


---

Table 1: Example of expanding MCDC tests to be RORG adequate

<table>
<thead>
<tr>
<th>Test Value</th>
<th>( t_1 )</th>
<th>( t_2 )</th>
<th>( t_3 )</th>
<th>( t_4 )</th>
<th>New ((t_1))</th>
<th>New ((t_2))</th>
<th>New ((t_3))</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a \neq b )</td>
<td>TTT</td>
<td>TTT</td>
<td>TTT</td>
<td>TTT</td>
<td>TTT</td>
<td>TTT</td>
<td>TTT</td>
</tr>
<tr>
<td>( a = b )</td>
<td>TTT</td>
<td>TTT</td>
<td>TTT</td>
<td>TTT</td>
<td>TTT</td>
<td>TTT</td>
<td>TTT</td>
</tr>
<tr>
<td>( a &lt; b )</td>
<td>TTT</td>
<td>TTT</td>
<td>TTT</td>
<td>TTT</td>
<td>TTT</td>
<td>TTT</td>
<td>TTT</td>
</tr>
<tr>
<td>( a &gt; b )</td>
<td>TTT</td>
<td>TTT</td>
<td>TTT</td>
<td>TTT</td>
<td>TTT</td>
<td>TTT</td>
<td>TTT</td>
</tr>
</tbody>
</table>


