Structural Equivalence Partition and Boundary Testing

Norbert Oster and Michael Philippsen
Computer Science Department, Programming Systems Group
University of Erlangen, Germany
[oster, philippsen]@cs.fau.de

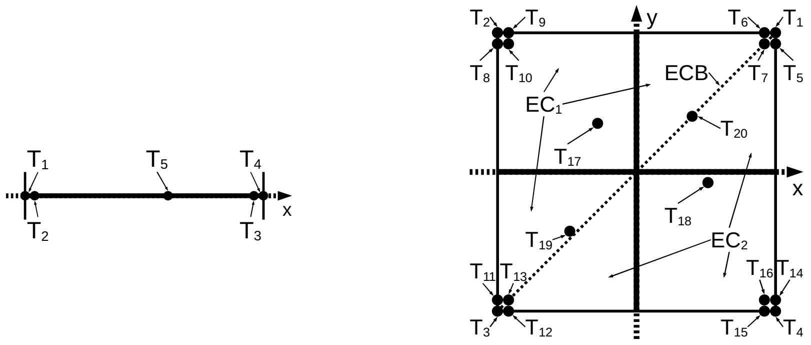
Abstract: Structural (manual or automated) testing today often overlooks typical programming faults because of inherent flaws in the simple criteria applied (e.g. branch or all-uses). Dedicated testing strategies that address such faults (e.g. mutation testing) are not specifically designed for smart automatic test case generation. In this paper we present a new coverage criterion and its implementation that accomplishes both: it detects more faults and integrates easily into automated test case generation. The criterion is targeted towards unveiling faults that originate from shifts in the equivalence classes that are caused by small coding errors (inspired by mutation testing). On benchmark codes from the Java-API and from an open-source project we improve the fault detection capability by up to 41% compared to branch and all-use coverage.

1 Introduction

Testing is still important in modern software engineering. Although remarkable progress has been made in the field of white-box (structural) testing, the average fault detection capability achieved with traditional coverage criteria (e.g. branch or all-uses) is still too low [Bis02, MMB03, Ost07], even if test case generators are used to produce test sets with high coverage ratios. The reason is that although programs often fail at the boundary of processing domains (e.g. due to primitive typo-like faults such as a < instead of a <= in a conditional expression), test cases designed for branch or all-uses coverage typically fail to detect such boundary faults. The test cases are *somewhere* on both sides of the boundary but not *at* the boundary. Even dataflow oriented testing (all-uses coverage) fails because it just checks the flow of information through the program withoug considering the *values* processed.

Mutation analysis (MA) is a better way to tell whether a test set is likely to find many bugs in the system under test (SUT), even *at* the borderline mentioned above [Lig02, OMK04]. For each test case MA compares the behavior of the SUT with that of its mutant, i.e. a copy with an artificially introduced fault. The ratio of mutants that show altered behavior under a test is called *mutation score* (MS) and is an objective indicator of the fault detection capability of this test set. The problem is that MA is not constructive, i.e. so far the idea of MA cannot be directly used to guide the automatic generation of good test cases.

In this paper we present a new coverage criterion called *S̲tructural E̲quivalence Partition and B̲oundary T̲esting* (SEBT) that implements the insights of MA and that *can* be easily used in test case generators to produce better test sets than with traditional criteria.



(a) `x++`: unary, one dimension (b) `x < y`: binary, two dimensions

Figure 1: Unary and binary operator examples.

The paper is structured as follows. Section 2 introduces our new criterion SEBT. Section 3 describes the tool support available for SEBT. Section 4 shows the results we achieve with this approach. Section 5 discusses related approaches before the paper concludes in section 6 and gives an outlook on our future plans.

2 Structural equivalence partition and boundary criterion

In general, languages like JAVA provide unary (e.g. post-increment `a++` or boolean negation `!b`), binary (e.g. division `a/b` or logical "and" `a&&b`), and ternary (e.g. shortcut for value-returning if-then-else `a ? b : c`) operators. We further distinguish four operand classes according to their data type: *enumerable* with constant values (e.g. `boolean` with `true` and `false`); *discrete*, where neighboring elements always have a constant distance of 1 (e.g. `byte`, `char`, `short`, `int`, `long`); *pseudo-real*, where the values are discrete due to limited machine representation but the distances between the elements vary (e.g. `float`, `double`); and finally *references* to objects or `null`.

In the following, we will focus on discrete and pseudo-real operands of unary and binary operators. Since `a ? b : c` is the only ternary operator in JAVA and has special typing requirements, we discuss it on its own. In this paper we do not address reference type operands, as this topic is out of the scope of SEBT yet. Whenever at least one of the operands of an operator under consideration is `enum`erable, we require testing the corresponding statement with all possible values of that operand in combination with the classes of the other operands, as described next.

Fig. 1 shows two examples of the distribution of the equivalence classes and their boundaries for an unary and a binary operator. Since the expression `x++` depends on only one variable, its input domain in terms of testing is the whole domain of `x`'s data type. Its input domain is regarded as one equivalence class. If tested on its own, the expression should be evaluated with different values of `x`, covering both the equivalence class and its boundaries, as shown in Fig. 1(a). Test cases T_1 and T_4 represent the limits of the data type of `x`, similar to the *on-points* in [WC80]. For `double` in JAVA, T_4 means

`Double.MAX_VALUE` $\approx 2^{1024}$. Respectively, T_2 and T_3 are seen as the near-boundaries of the class, similar to the *off-points* in [WC80], and should be tested as well. The values to be chosen here for `x` should be those immediately next to T_1 resp. T_4 in terms of machine arithmetics. For practical applicability, the tester may provide a distance δ such that any value $T_2 \in]T_1, T_1 + \delta]$ and $T_3 \in [T_4 - \delta, T_4[$ would be acceptable. Finally, an arbitrary representant $T_5 \in]T_1 + \delta, T_4 - \delta[$ within the equivalence class should be tested as well.

Two equivalence classes and twenty test cases are reasonable for checking the expression `x < y`, as shown in Fig. 1(b). The dotted diagonal line represents the boundary of the two equivalence classes for the given expression: equivalence class EC_1 comprises all pairs (x, y) such that the condition `x < y` holds, while EC_2 represents the opposite. Test cases $T_1, T_2, \ldots, T_{16}$ represent the boundary or near-boundary values of the data types of both variables. Additionally, arbitrary inner representants (here: T_{17} and T_{18}) from within each class are required, that are not a (near-)boundary input at the same time. The statement `x < y` must be tested with near-boundary values on "both sides" of each equivalence class as well. In Fig. 1(b), the latter is achieved by the test cases T_{19} and T_{20} – where T_{20} represents pairs (x, y) satisfying the condition $x = y$ and thus the boundary of EC_2 towards EC_1 (a generic formal definition of both is given in section 2.1). In contrast to [WC80], we do not distinguish *on-points* and *off-points* for the boundary between classes, since we expect boundary and near-boundary test cases for all classes, thus implicitly always requiring both kinds of "points".

In the case of binary operators, we will distinguish between relational (i.e. <, <=, =, !=, >=, >) and arithmetic (i.e. `+, -, *, /, %, ^, ...`) expressions. For relational operators we require adequate test cases such that the operands are evaluated to values identical to or immediately at the limits imposed by their data type, e.g. $T_1 - T_{16}$ in Fig. 1(b). Arithmetic operators put additional restrictions on the operands, that might be of higher interest for testing: the boundaries of the *result*. As an example, consider a simple addition like `x + y`. In the visualization of this (binary) operator according to Fig. 1(b), test cases T_1 and $T_5 - T_7$ (respectively T_3 and $T_{11} - T_{13}$) will trigger overflows (resp. underflows). Thus, new boundaries for reasonable equivalence classes are imposed by the conditions $x + y < t^{x,y}_{min}$ or $x + y > t^{x,y}_{max}$, where $t^{x,y}_{min}$ and $t^{x,y}_{max}$ are the boundaries of the resulting data type after implicit conversion [GJSB05].

2.1 Generic SEBT criterion

Based on the examples above we can now formally define the SEBT criterion.

Unary operators: Let *op* be an unary operator applied to an expression (e.g. variable) v of type t^v and $\overline{op} \neq op$ a type-compatible operator, which can syntactically replace *op*. Further let $\delta^v > 0$ be the minimum distance between the lowest possible value t^v_{min} of v according to t^v and the smallest value $t^v_{min} + \delta^v$, such that $]t^v_{min}, t^v_{min} + \delta^v[= \emptyset$; respectively let $\epsilon^v > 0$ be the minimum distance between the highest possible value t^v_{max} of v according to t^v and the greatest value $t^v_{max} - \epsilon^v$, such that $]t^v_{max} - \epsilon^v, t^v_{max}[= \emptyset$. For practical feasibility, δ^v and ϵ^v may be relaxed to arbitrary but adequate distances chosen by the tester in accordance with the SUT.

SEBT requires that for a given statement v *op* or *op* v, the expression must be executed at least once with each of the following test cases: $v_1 = t^v_{min}$, $v_2 \in]t^v_{min}, t^v_{min} + \delta^v]$, $v_3 \in [t^v_{max} - \epsilon^v, t^v_{max}[$, $v_4 = t^v_{max}$, and for each compatible $\overline{op}$ with at least one test case $v_5 \in]t^v_{min} + \delta^v, t^v_{max} - \epsilon[$ such that the expression behaves differently (e.g. for arithmetic expressions: *op* $v_5 \neq \overline{op}\, v_5$). Example: `-1` $\neq$ `+1` (but `-0` vs. `+0` is *not* enough).

Binary operators: Let *op* be a binary operator applied to two expressions (e.g. variables) a of type t^a respectively b of type t^b returning a result of type $t^{a,b}$ and $\overline{op} \neq op$ a type-compatible operator, which can syntactically replace *op*. Further let δ^a, δ^b, $\delta^{a,b}$, t^a_{min}, t^b_{min}, $t^{a,b}_{min}$, ϵ^a, ϵ^b, $\epsilon^{a,b}$, t^a_{max}, t^b_{max}, $t^{a,b}_{max}$ have the same meaning for a, b, and the result as in the definition above for v in unary expressions. Additionally, let $\xi^a_{(a)}$ ($\xi^b_{(b)}$) be the smallest distance between the current value of variable a (b) and the values immediately next to it according to the machine precision with respect to the data type t^a (t^b) – please note that in JAVA, ξ is *not* necessarily constant, e.g. $\xi^{double}_{(x)} \in [5 \cdot 10^{-324}, 10^{292}]$.

SEBT requires that for a given statement a *op* b and each compatible $\overline{op}$, the expression must be executed with at least one test case from each of the generic test classes determined by the conditions in Table 1 (upper and middle part), which is based on abbreviations defined in Table 2, distinguishing between relational and arithmetic operators. For the example `x < y`, twenty test cases satisfying SEBT are depicted graphically in Fig. 1(b).

Ternary operators: The conditional operator `a ? b : c` in JAVA requires `a` to be of type `boolean`. A coding error may only result from interchanging the three operands, if all three are of the same type – or at least the second and third operand, which must be of compatible type anyway. In order to detect such programming faults, we take advantage of the idea of *modified condition/decision coverage (MC/DC)* [Lig02], as described in the following. Let t be an arbitrary test case and $v_t(a)$ (respectively $v_t(b)$ and $v_t(c)$) be the (hypothetical) evaluation result of operand `a` (resp. `b` and `c`) when executing the statement under test with test case t. Please note that due to short-cut evaluation in JAVA, at least one of the operands is not evaluated each time the expression is executed.

SEBT requires that for a given statement `a ? b : c`, the test set must contain at least one pair (t_1, t_2) of test cases for each of the generic test classes in Table 1 (lower part).

2.2 Special consideration: Feasibility

When executing program logics, *feasibility* may prevent certain structural entities (e.g. def/use-pairs) from being covered, although required by the chosen criterion. Often the reason is that no input (no test case) can be found, that reaches certain nodes or edges of the control flow graph. Depending on the SUT, the generic SEBT criterion defined above may also be affected by such infeasibility for certain test classes. For example, consider the pair *op* := "<" and $\overline{op}$:= ">=" of arithmetic operators. No input exists that covers the generic test classes T_{17}, T_{19}, and T_{20}, because the equivalence class EC_1 is empty, i.e. $\nexists x, y : x{<}y \wedge x{>=}y$. Any tool implementation of SEBT must account for such combinations to compute a reasonable coverage measure.

Table 1: Generic test classes (See Table 2 for the legend).

	Class	*Description*	*Conditions on operands*
for ***binary relational*** *operators*	T_1	limit (I)	$a = t^a_{max} \wedge b = t^b_{max}$
	T_2	limit (II)	$a = t^a_{min} \wedge b = t^b_{max}$
	T_3	limit (III)	$a = t^a_{min} \wedge b = t^b_{min}$
	T_4	limit (IV)	$a = t^a_{max} \wedge b = t^b_{min}$
	T_5	near-limit (I)	$a = t^a_{max} \wedge b \in]t^b_{max}, t^b_{max} - \epsilon^b]$
	T_6	near-limit (I)	$a \in]t^a_{max}, t^a_{max} - \epsilon^a] \wedge b = t^b_{max}$
	T_7	near-limit (I)	$a \in]t^a_{max}, t^a_{max} - \epsilon^a] \wedge b \in]t^b_{max}, t^b_{max} - \epsilon^b]$
	T_8	near-limit (II)	$a = t^a_{min} \wedge b \in]t^b_{max}, t^b_{max} - \epsilon^b]$
	T_9	near-limit (II)	$a \in]t^a_{min}, t^a_{min} + \delta^a] \wedge b = t^b_{max}$
	T_{10}	near-limit (II)	$a \in]t^a_{min}, t^a_{min} + \delta^a] \wedge b \in]t^b_{max}, t^b_{max} - \epsilon^b]$
	T_{11}	near-limit (III)	$a = t^a_{min} \wedge b \in]t^b_{min}, t^b_{min} + \delta^b]$
	T_{12}	near-limit (III)	$a \in]t^a_{min}, t^a_{min} + \delta^a] \wedge b = t^b_{min}$
	T_{13}	near-limit (III)	$a \in]t^a_{min}, t^a_{min} + \delta^a] \wedge b \in]t^b_{min}, t^b_{min} + \delta^b]$
	T_{14}	near-limit (IV)	$a = t^a_{max} \wedge b \in]t^b_{min}, t^b_{min} + \delta^b]$
	T_{15}	near-limit (IV)	$a \in]t^a_{max}, t^a_{max} - \epsilon^a] \wedge b = t^b_{min}$
	T_{16}	near-limit (IV)	$a \in]t^a_{max}, t^a_{max} - \epsilon^a] \wedge b \in]t^b_{min}, t^b_{min} + \delta^b]$
	T_{17}	representant (EC_1)	$C_0 \wedge C_1 \wedge C_2$
	T_{18}	representant (EC_2)	$C_0 \wedge C_3 \wedge C_4$
	T_{19}	boundary (EC_1)	$C_0 \wedge C_1 \wedge \neg C_2$
	T_{20}	boundary (EC_2)	$C_0 \wedge C_3 \wedge \neg C_4$
for ***binary arithmetic*** *operators*	T_1	underflow (I)	$a\ op\ b < t^{a,b}_{min} - \delta^{a,b}$
	T_2	overflow (II)	$a\ op\ b > t^{a,b}_{max} + \epsilon^{a,b}$
	T_3	near-limit (Ia)	$a\ op\ b \in [t^{a,b}_{min} - \delta^{a,b}, t^{a,b}_{min}[$
	T_4	near-limit (IIa)	$a\ op\ b \in]t^{a,b}_{max}, t^{a,b}_{max} + \epsilon^{a,b}]$
	T_5	limit (I)	$a\ op\ b = t^{a,b}_{min}$
	T_6	limit (II)	$a\ op\ b = t^{a,b}_{max}$
	T_7	near-limit (Ib)	$a\ op\ b \in]t^{a,b}_{min}, t^{a,b}_{min} + \delta^{a,b}]$
	T_8	near-limit (IIb)	$a\ op\ b \in [t^{a,b}_{max} - \epsilon^{a,b}, t^{a,b}_{max}[$
	T_9	representant (EC_1)	$C_0 \wedge C_1 \wedge C_2 \wedge C_5$
	T_{10}	representant (EC_2)	$C_0 \wedge C_3 \wedge C_4 \wedge C_5$
	T_{11}	boundary (EC_1)	$C_0 \wedge C_1 \wedge \neg C_2 \wedge C_5$
	T_{12}	boundary (EC_2)	$C_0 \wedge C_3 \wedge \neg C_4 \wedge C_5$
ternary	T_1	`a` true, vary `b` only	$a = true \wedge v_{t_1}(b) \neq v_{t_2}(b) \wedge v_{t_1}(c) = v_{t_2}(c)$
	T_2	`a` false, vary `c` only	$a = false \wedge v_{t_1}(b) = v_{t_2}(b) \wedge v_{t_1}(c) \neq v_{t_2}(c)$
	T_3	vary `a`, keep `b` and `c` with `b` $\neq$ `c`	$v_{t_1}(a) \neq v_{t_2}(a) \wedge v_{t_1}(b) \neq v_{t_1}(c) \wedge$ $v_{t_1}(b) = v_{t_2}(b) \wedge v_{t_1}(c) = v_{t_2}(c)$

Table 2: Abbreviations of conditions used in Table 1.

Abbrev.	Description	Condition on operands
C_0	non-limit	$a \in]t^a_{min} + \delta^a, t^a_{max} - \epsilon^a[\wedge b \in]t^b_{min} + \delta^b, t^b_{max} - \epsilon^b]$
C_1	same behavior	$a\ op\ b = a\ \overline{op}\ b$
C_2	same behavior (at neighborhood)	$\forall \psi^a \in \{-\xi^a_{(a)}, 0, \xi^a_{(a)}\}, \forall \psi^b \in \{-\xi^b_{(b)}, 0, \xi^b_{(b)}\}$: $(a + \psi^a)\ op\ (b + \psi^b) = (a + \psi^a)\ \overline{op}\ (b + \psi^b)$
C_3	different behavior	$a\ op\ b \neq a\ \overline{op}\ b$
C_4	different behavior (at neighborhood)	$\forall \psi^a \in \{-\xi^a_{(a)}, 0, \xi^a_{(a)}\}, \forall \psi^b \in \{-\xi^b_{(b)}, 0, \xi^b_{(b)}\}$: $(a + \psi^a)\ op\ (b + \psi^b) \neq (a + \psi^a)\ \overline{op}\ (b + \psi^b)$
C_5	non-under/overflow (non-limit)	$a\ op\ b \in]t^{a,b}_{min} + \delta^{a,b}, t^{a,b}_{max} - \epsilon^{a,b}[\wedge$ $a\ \overline{op}\ b \in]t^{a,b}_{min} + \delta^{a,b}, t^{a,b}_{max} - \epsilon^{a,b}[$

3 Tool support

Regardless of the testing strategy, software testing today is hardly possible without adequate tool support. There is a broad variety of approaches to automatic test case generation. Random test data generators produce myriads of redundant test cases, covering the same entities of the control or data flow several times [MMS98, BHJT00]. Other techniques try to remove redundant test cases *after* their random generation, thus giving smaller test sets [OW91, Ton04, McM04]. But the general problem remains NP-complete.

Heuristic test case generators [WBP02, OS06, PBSO08, FZ10] are usually driven by one or more *objective functions*. Those objectives compute quantifiable characteristics such as the structural coverage achieved (e.g. according to the branch criterion) or the "distance" [Bar00] of an individual test case from covering a certain test goal (e.g. a statement not yet executed). In general, such a generator first instruments the SUT once. It then randomly generates a set of test cases. Each test case is executed and evaluated according to each objective. The results of the evaluation are then used to guide the heuristics towards generating new and better test cases (or test sets) based on the old ones. The cycle of generation, execution, and evaluation is repeated until the test set is considered good enough.

An important characteristic of SEBT is that this coverage criterion can be implemented as such an objective function plug-in for arbitrary heuristic test case generators. Hence SEBT makes it possible to generate test cases that achieve better mutation scores than traditional coverage criteria and that hence help build more trust in the test. We have implemented the SEBT criterion for JAVA as such a plug-in. It provides hooks for *instrumentation*, *execution*, and *evaluation*. The three hooks of the SEBT plug-in work as follows.

Instrumentation: We parse the source code and construct an abstract syntax tree. In a transformation we then insert so-called probes into the given source code that do not modify the semantics of the original program. The probes are calls to a logging subsystem. The instrumentation tool logs each transformation in a so-called Static Logging Data (SLD) file that contains information on the instrumented statements: a unique identifier and the kind of the occurring operator. As an example, consider an excerpt from the `Dijkstra` benchmark (section 4):

```
while (nodesToProcess.size() > 0) {
    Node nextNodeToProcess = getNodeWithSmallestDistance();
    nodesToProcess.remove(nextNodeToProcess);
    Vector neighbours = nextNodeToProcess.getNeighbours();
    for (int i = 0; i < neighbours.size(); i++) {
        Node neighbour = (Node)neighbours.get(i);
        if (neighbour.getCostFromRoot() >
                nextNodeToProcess.getCostFromRoot()
                + nextNodeToProcess.getCostToNeighbour(neighbour)) {
            neighbour.setPredecessor(nextNodeToProcess);
```

The automatic instrumentation gives the following (pretty-printed) code:

```
while (logger.my_gt_function("12_1_4", nodesToProcess.size(), 0)) {
    Node nextNodeToProcess = getNodeWithSmallestDistance();
    nodesToProcess.remove(nextNodeToProcess);
    Vector neighbours = nextNodeToProcess.getNeighbours();
    for(int i=0; logger.my_lt_function("12_1_5", i, neighbours.size());
            i = logger.my_post_inc_function ("12_1_6", i)) {
        Node neighbour = (Node)neighbours.get(i);
        if (logger.my_gt_function("12_1_8", neighbour.getCostFromRoot(),
                logger.my_plus_function ("12_1_7" ,
                    nextNodeToProcess.getCostFromRoot(),
                    nextNodeToProcess.getCostToNeighbour(neighbour)))) {
            neighbour.setPredecessor(nextNodeToProcess);
```

Furthermore, it adds corresponding entries to the SLD file, shown in Fig. 2(left).

```
                    | 12_1_8;GT;double;double;1.797...157E308;1.797...157E308
12_1_4;GT           | 12_1_8;GT;double;double;1.797...157E308;2.0
12_1_5;LT           | 12_1_8;GT;double;double;2.0;2.0
12_1_6;POST_INC     | 12_1_8;GT;double;double;5.0;1.797...157E308
12_1_7;PLUS         | 12_1_8;GT;double;double;5.0;10.0
12_1_8;GT           | 12_1_8;GT;double;double;0.0;4.9E-324
```

Figure 2: Excerpts from the SLD (left) and DLD files of the `Dijkstra` example.

Although we did not execute any test case yet, the plug-in derives from statically analyzing those lines of the SLD, that the code excerpt contains two occurrences of the operator ">" (GT). Since we must cover 20 test case classes (see Table 1) for each feasible pair of op := ">" and any compatible $\overline{op}$, this leads to well-defined 74 different operand combinations, in order to fully satisfy SEBT for the two probes with the IDs `12_1_4` and `12_1_8`. Of course, the other 41 IDs in the SLD of the entire code must be considered accordingly.

Execution: The transformed code is executed for each test case. This results in a Dynamic Logging Data (DLD) file for each test run. Whenever a probe is executed, the unique ID, the type of the operator, the runtime types, and the actual values of each evaluated operand are logged. An excerpt from a DLD, containing some of the essential entries for the operator ID `12_1_8` is shown in Fig. 2(right).

Evaluation: Finally, all current SLDs and DLDs are considered in order to determine the coverage ratio achieved. Even if some relevant statements remain totally uncovered by the test set (and thus do not show up in the DLD), the SLD provides the necessary information to account for them as well. From the runtime types of the operands in the DLD the tool can determine the limits t_{min} and t_{max} of the data-types of each individual operand (see section 2.1, Table 1). Merging the knowledge from the above SLD and DLD excerpts for ID `12_1_8`, we notice that $t^a_{max} = t^b_{max} = t^{double}_{max} = 1.797\ldots157E308$ and $\xi^{double}_{(0)} = 4.9E-324$. Among others, we covered the test classes T_1 (1st line of DLD); T_{20} (3rd line of DLD), T_{17} (5th line), and T_{19} (6th line) for op := ">" vs. $\overline{op}$:= ">="; as well as many other test classes for further compatible $\overline{op}$ at the same time.

From this evaluation, the plug-in reports to the driving heuristic test case generator which SEBT classes T^u_i have not been covered yet. Moreover, for each pair of test case t_j and test class T^u_i, the plug-in also provides a means to compute the "distance" of t_j from covering T^u_i, based on a distance metrics from graph theory applied to the control flow graph plus a set of distance functions for conditional expressions applied to branch predicates (see [Bar00] for details). The test case generator can than choose the test class T^u_s with the smallest distance as the new test target and can iteratively evolve an adequate subset of the current population of test cases t_j towards covering T^u_s. In case of a test generator that is based on a genetic algorithm, the distance objective can serve as the fitness value for the selection operator. If T^u_s can be covered by a test case t^s_j within a predefined number of steps, then t^s_j is added to the set of resulting test cases. This is done for all test targets separately.

In the above example, the DLD does not contain an entry covering T_2, but the nearest test case achieves operand values of `5.0;1.797...157E308`. The heuristics is now guided to evolve the current value of the first operand (`5.0`) towards the smallest possible value for the datatype `double`, thus covering T_2.

Table 3: Benchmark results

Project	*# of Java classes*	*lines of code*	*source size (bytes)*	*# of* mutants	*number of killed mutants and mutation scores in %*						*improvement*	
					branch (B)		*all-uses* (A)		*SEBT* (S)		*B→S*	*A→S*
Hanoi	1	38	1279	227	170	75%	174	77%	197	87%	27	23
JDKsort	1	82	2639	852	524	62%	557	65%	577	68%	53	20
Dijkstra	2	141	4080	220	155	70%	158	72%	206	94%	51	48
Huffman	2	298	8931	623	390	63%	390	63%	391	63%	1	1
BigFloat	3	540	17526	1528	1057	69%	1159	76%	1494	98%	437	335
					provided JUnit tests (PJT)				*PJT+SEBT*		*by SEBT*	
JTopas	44	16112	583546	3219	1518			47.2%	1854	57.6%		336

improvement : Number of faults detected with SEBT but *missed* with branch/all-uses/provided JUnit test cases

Targeting each test class on its own results in intractably large test sets, since such approaches strive for maximized coverage without considering the validation effort in terms of the resulting total number of test cases. SEBT can also be used to guide test case generators that do take the size of the test set into account, e.g. by adding another global optimization phase to the generation cycle described above. For instance, in its global optimization phase, our tool •gEAr [Ost07] works on a *set of test sets*, i.e. not just on a single test set. Each of these test sets is evaluated with respect to its size and the coverage ratio computed by the SEBT plug-in. On these weighted test sets •gEAr applies a genetic algorithm (with cross-over between test sets and mutation within test sets) to construct smaller test sets with similar (or better) SEBT-coverage. Only if this global optimization of the test sets cannot rationally improve the SEBT-coverage or the test set size anymore, i.e., if for example a certain test class T_s^u remains uncovered within an upper number of iterations, •gEAr uses the mechanism described above to evolve those existing test cases that are close with respect to SEBT towards covering T_s^u. The SEBT-plug-in is hence used in both phases that •gEAr repeats iteratively until a test set is found that not just has a good coverage with respect to SEBT but that is also small.

4 Experimental results

To evaluate the power of SEBT, we conducted several experiments in two different setups. The characteristics of the SUTs used and the results achieved are shown in Table 3. Mutation analysis is more adequate to assess the quality of a test set in terms of its fault detection capability, than the mere coverage measure. Killing a mutant means that the corresponding test case is sensitive to and thus able to detect a certain (potential) fault. Additionally, MA is independent of the coverage criterion actually applied to derive the test set. For each SUT, we automatically generated mutants with MuJava [OMK04], applying *all* mutation operators provided by the tool (including class mutation). These sets of mutants may contain functionally equivalent programs.

Setup A (upper part of Table 3): The SUTs used in this setup are textbook examples, such as Dijkstras shortest path algorithm. `JDKsort` is taken from the JAVA-API. `BigFloat` is a simplified variant of java.math.BigDecimal. We generated three test sets for each SUT, one achieving 100% branch coverage, the second satisfying the all-uses criterion. The third test set has covered the feasible test classes of SEBT for relational and arithmetic operators. For all three groups of test sets, we determined the number of mutants killed. Although the all-uses criterion is quite demanding, the average fault detection (represented by the mutation score) achieved in setup A is about 70.7%; test sets satisfying branch coverage were even weaker and reach approx. 66.6%. The test sets for SEBT performed best and detected an average of 83.0% of the potential faults.

In our benchmarks, we apply SEBT for arithmetic and relational statements. Thus the criterion performs best on SUTs whose behavior mainly relies on such operators and the pure test values, i.e., control and data flow strongly depend on the actual input, rather than on the internal state with weak or no input dependence. Because the `Huffman` algorithm represents straightforward encoding, the input values themselves are not directly processed by arithmetic or relational operations. Instead of the input values, the number of occurrences of different symbols in the input is computed and used to guide the control flow. Since the establishing of almost all SEBT-relevant data is infeasible (e.g. the total input length must always be a small but positive integer due to memory limitations and the JAVA-API specification), SEBT killed only one additional mutant – branch and all-uses perform equally bad. For `BigFloat` however, SEBT has killed 437 (+29%) mutants, that were missed by branch coverage. This is due to the intense number crunching of `BigFloat`, that applies arithmetic and relational operators immediately to the input values. Since SEBT can execute the relevant statements with almost all operand configurations, the resulting fault detection of 98% is impressive.

Setup B (lower part of Table 3): The SUT in this setup is the entire “Java tokenizer and parser tools” package `JTopas` (`http://jtopas.sourceforge.net/jtopas`), comprising 44 JAVA classes with currently 16112 lines of open source code. The `JTopas` distribution comprises a set of 552 JUnit test cases. We supplemented this given set with 584 additional JUnit test cases, that cover the test classes of SEBT.

The JUnit test cases provided with `JTopas` killed 1518 (47.2%) out of 3219 mutants, while the improved test set achieved a mutation score of 1854 (57.6%) – i.e. a total of 336 (+10.4%) mutants, that were *not* detected by the original JUnit tests, are now killed by the SEBT test cases. Hence, SEBT significantly improves the chance of fault detection in an open source project with existing JUnit test cases.

5 Related work

Many different testing strategies and test automation tools exist, but there are very few approaches to automatic test data generation for more demanding testing criteria, and almost all are academic. Test data for structural coverage is typically derived by applying so-called program slicing [FB97].

Static slicing [WC80, KLPU04] explicitly enumerates all control or data flow paths required (e.g. directly for path coverage or indirectly for statement coverage), i.e. all constraints imposed by the conditions at the branching nodes are collected for each path in part. The resulting constraint system in terms of the input variables represents the domain partitioning of the program with respect to individual control flow paths. Compared to the correct version of the code, faults in the program are expected to either shift some "domain boundary" of the solved constraint system (follow wrong branch for a certain input) or modify some "domain computation" (give wrong output despite following the correct path). Since statically setting up the corresponding constraint system is very expensive or even infeasible (for loops the number of paths is infinite), those approaches have many restrictions. White and Cohen [WC80] introduce their technique for a simplified Algol-like programming language that lacks many features of modern object-oriented languages. Moreover, they make restrictive assumptions with respect to the types of constraints and the input space (only linear borders between domains are allowed). Kosmatov, Legeard et al. [KLPU04] apply the concept to formal models of the SUT. In case of state machines, the predicate conditions (i.e. guards in state transitions) are collected to define the input domains for each modeled behavior. Their approach applies to discrete domains for (simple) formal model languages only, and must also fail for systems with unknown numbers of loop iterations. They do not address source code testing at all. In contrast to the techniques presented above our SEBT criterion is intended for arbitrary imperative or object-oriented languages, and the prototypical implementation for JAVA does not exclude any language feature. In order to cope with the limits of static analysis and to ease the automatic generation of corresponding test cases, SEBT allows generators to consider individual code statements only instead of reckoning complete program paths. Inspired by the concept of *weak* mutation testing [OMK04], SEBT is satisfied with test cases that execute individual statements with manifold data contexts, able to trigger possibly faulty behavior at each program location (e.g. < vs. <=) in part.

In general, dynamic program slicing [Bar00, WBP02, FZ10] executes the SUT with a random test case first. If the desired path has been traversed, the corresponding input is directly available. Otherwise, some heuristics are applied to push the currently covered path towards the desired one, usually by repeatedly applying gentle modifications to the best input known so far. Many approaches of this kind start with a static control flow analysis in order to identify the target entities to be covered according to the coverage criterion. Such techniques suffer from the same drawbacks as static slicing (e.g. infinite number of paths) and are therefore only applicable to simple criteria based on control flow such as branch coverage. Our new criterion SEBT can serve as a driving plug-in for the approach by Wegener et al. [WBP02], but as mentioned above targeting each entity in part will likely result in intractably huge test sets for average programs.

Other techniques collect the test targets "on the fly" during the heuristic optimization and explicitly target missed entities without prior analysis of the entire control flow. Fraser and Zeller [FZ10] reduce the testing effort by minimizing the length of each test case, i.e. roughly the number of statements in the test sequence. In contrast, with SEBT in •gEAr the tester can select between short but rather many test cases similar to [FZ10] and a minimized number of test cases, i.e. a small test set. In the latter case, each test run covers several combinations of SEBT classes at once. Moreover, [FZ10] uses a sophisticated impact analysis of each mutant execution, in order to assess to which extent a mutant has been covered: The more statements are covered in the original code but not in the mutant and vice versa and the more methods behave differently (i.e. return different values or throw unexpected exceptions) in the mutant compared to the original code, the higher is the impact of this mutant under an assessed test case. If a test case does not cause enough impact of the mutant under assessment (e.g. due to failure masking in the early testing stages) the test case is quite likely dismissed during the generation of the final test set. Nevertheless, such a test case may become important during regression testing. In contrast, SEBT requires to initially cover all operand-value-combinations expected to cause failures (sooner or later).

6 Conclusion and outlook

In this paper, we presented a new testing technique named *structural equivalence partition and boundary testing (SEBT)* based on the notion of mutation testing. The goal of SEBT is to help the tester to systematically choose test cases sensitive to typical coding errors, such as unintentionally using the wrong operator (e.g. "$<$" instead of "$\leq$") or not accounting for limits (over-/underflow) and limitations (machine precision) of the data-types used. The tool computes the coverage achieved by a given test set for the code portion under test according to SEBT. Experimental results are very promising: SEBT outperforms branch coverage by 41% and all-uses coverage by 30% (w.r.t. the mutation score achieved).

The SEBT criterion is primarily meant for classical operators like arithmetical $(+, -, *, /)$ and logical (&&, $||$, !) ones. Nevertheless, the key idea can easily be extended to more sophisticated language features. We already examined its applicability to coding errors concerning arrays, general computation failures due to rounding, inheritance, polymorphism, method overloading, and to data flow aspects [Fri09]. Applying SEBT to multi-threading faults is still an open and challenging task.

References

[Bar00] A. Baresel. Automatisierung von Strukturtests mit evolutionären Algorithmen. Diploma thesis, FG Softwaretechnik, Humboldt-Universität Berlin, July 2000.

[BHJT00] B. Baudry, Vu Le Hanh, J.-M. Jézéquel, and Y. Le Traon. Building Trust into OO Components Using a Genetic Analogy. In *ISSRE 2000: Proc. 11th Intl. Symp. Software Reliability Engineering*, pages 4–14, Washington, DC, Oct. 2000.

[Bis02] P. G. Bishop. Estimating Residual Faults from Code Coverage. In *21st Intl. Conf. Computer Safety, Reliability and Security*, volume 2434 of *LNCS*, pages 163–174, Catania, Italy, Sep. 2002.

[FB97] I. Forgács and A. Bertolino. Feasible test path selection by principal slicing. *SIGSOFT Software Engineering Notes*, 22(6):378–394, Nov. 1997.

[Fri09] S. Fritz. Konzeption und Implementierung eines Verfahrens zur strukturellen Äquivalenzklassen- und Grenzwerttestüberdeckung. Diploma thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, CS Dept., Feb. 2009.

[FZ10] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. In *ISSTA '10: Proc. 2010 ACM SIGSOFT Intl. Symp. Software Testing and Analysis*, pages 147–158, Trento, Italy, July 2010.

[GJSB05] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification*, chapter Conversions and Promotions. Addison Wesley, 3 edition, June 2005.

[KLPU04] N. Kosmatov, B. Legeard, F. Peureux, and M. Utting. Boundary Coverage Criteria for Test Generation from Formal Models. In *ISSRE '04: Proc. 15th Intl. Symp. Software Reliability Engineering*, pages 139–150, Saint-Malo, France, Nov. 2004.

[Lig02] P. Liggesmeyer. *Software-Qualität – Testen, Analysieren und Verifizieren von Software*. Spektrum - Akademischer Verlag, Heidelberg; Berlin, 2002.

[McM04] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, June 2004.

[MMB03] M. M. Marré and A. Bertolino. Using spanning sets for coverage testing. *IEEE Trans. Software Engineering*, 29(11):974–984, Nov. 2003.

[MMS98] G. McGraw, C. Michael, and M. Schatz. Generating Software Test Data by Evolution. Technical Report RSTR-018-97-01, RST Corporation, Sterling, VA, Feb. 1998.

[OMK04] J. A. Offutt, YuSeung Ma, and YongRae Kwon. An Experimental Mutation System for Java. *Proc. Workshop Empirical Research in Software Testing / ACM SIGSOFT Software Engineering Notes*, 29(5):1–4, Sep. 2004.

[OS06] N. Oster and F. Saglietti. Automatic Test Data Generation by Multi-Objective Optimisation. In *Computer Safety, Reliability and Security*, volume 4166 of *LNCS*, pages 426–438, Gdansk, Poland, Sep. 2006.

[Ost07] N. Oster. *Automatische Generierung optimaler struktureller Testdaten für objekt-orientierte Software mittels multi-objektiver Metaheuristiken*. PhD thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, CS Dept., Feb. 2007.

[OW91] T. J. Ostrand and E. J. Weyuker. Data Flow-Based Test Adequacy Analysis for Languages with Pointers. In *Proc. Symp. Testing, Analysis, and Verification*, pages 74–86, New York, Oct. 1991.

[PBSO08] F. Pinte, G. Baier, F. Saglietti, and N. Oster. Automatische Generierung optimaler modellbasierter Regressionstests. In *INFORMATIK 2008 - Beherrschbare Systeme dank Informatik*, volume P-133 of *LNI*, pages 193–198. Ges. f. Informatik, Sep. 2008.

[Ton04] P. Tonella. Evolutionary testing of classes. In *ISSTA '04: Proc. 2004 ACM SIGSOFT Intl. Symp. Software Testing and Analysis*, pages 119–128, New York, July 2004.

[WBP02] J. Wegener, K. Buhr, and H. Pohlheim. Automatic Test Data Generation For Structural Testing Of Embedded Software Systems By Evolutionary Testing. In *GECCO 2002: Proc. Genetic and Evolutionary Comp. Conf.*, pages 1233–1240, New York, July 2002.

[WC80] L. J. White and E. I. Cohen. A Domain Strategy for Computer Program Testing. *IEEE Trans. Software Engineering*, SE-6(3):247–257, May 1980.