On Redundant Mutants and Strong Mutation

Birgitta Lindström
University of Skövde
Skövde, Sweden
birgitta.lindstrom@his.se

András Márki
University of Skövde
Skövde, Sweden
andras.marki@his.se

***Abstract*—This study evaluates a theory of subsumption relations among mutants created by the ROR mutation operator, thus making most of these mutants redundant. A redundant mutant can be skipped during mutation analysis without decreasing the quality of the resulting test suite. This theory is interesting since mutation testing is computationally expensive and the theory states that the set of ROR mutants can be reduced by 57%. The reduced set of ROR mutants has therefore, been used in several recent studies. However, we provide proof that this theory do not hold for strong mutation and that part of the theory is incorrect. The theory itself has to our knowledge, never before been evaluated empirically.**

By finding counter examples, we prove that a test suite, which is 100% adequate for the non-redundant ROR mutants might not be 100% adequate for the mutants, which are supposed to be redundant. The subsumption relations do not hold for strong mutation. We have also proved that more than one top-level mutant can be detected by the same test. This should not be possible according to the theory. Hence, this part of the theory is incorrect, independent of strong or weak mutation.

Our findings are important since strong mutation is frequently used to evaluate test suites and testing criteria. Just as redundant mutants can give an overestimation of the mutation score for a test suite, using the reduced set can give an underestimation. Results reported from such studies should therefore, be accompanied by information on whether the reduced or complete set of ROR was used and if the researchers used strong or weak mutation.

***Index Terms*—mutation testing, redundant mutant, strong mutation**

I. Introduction

Mutation testing was originally introduced by DeMillo et al. [1]. In mutation testing, multiple (often faulty) versions of the software, *mutants* are created by systematically applying *mutation operators*, which are rules for changing syntactic elements. Tests are then designed to detect the mutants. A mutant is detected by a test if its behavior is different from the original (un-mutated) software. Mutation testing is often used to asses the quality of a test suite. The percentage of the mutants that a test suite detects is called the *mutation adequacy score*. This score is frequently used as a *golden standard*, against which other testing techniques are evaluated.

Mutation testing is an effective technique when it comes to detecting faults. Li et al. [2] performed a study in order to evaluate the cost-effectiveness of mutation. The authors computed the ratio of the size of the test suite and the number of faults that were detected in a case study comparing mutation, all-uses, edge-pair and prime path coverage. The authors showed that the ratio test/faults can be much lower for mutation testing than for the other techniques. This shows that mutation testing might be the best option when it comes to cost-effectiveness of the resulting test suite.

Mutation testing is however, known to be computationally expensive. This has to do with the large number of mutants generated. For example, mutation of source code typically generates $O(d^2)$ mutants where d is the number of data references [3]. Each of these mutants contributes to the cost since each mutant must be created, executed and analyzed.

Each mutant represents a test requirement, i.e., to design a test that detects this mutant. Hence, the number of such requirements is very large but the number of tests needed to fulfill these requirements is usually much lower than the number of mutants since one test often detects many mutants. A major reason for this is that many of the analyzed mutants are redundant. A *redundant mutant* contributes to the cost of the analysis but does not contribute to the overall quality of the resulting test suite.

The reason for why the redundant mutant does not contribute to the quality of the test suite is that a redundant mutant is guaranteed to be detected by the test suite if all non-redundant mutants are detected. The number of redundant mutants can be very high so there is a great potential to lower the cost for mutation testing if the redundant mutants can be identified at an early stage. Reducing the set of mutants to a minimal set of non-redundant mutants would therefore, make mutation testing much more attractive to industry.

Besides from the cost the redundant mutants introduce, they also obscure the results in empirical studies. The redundant mutants introduce a noise to the mutation adequacy score, which makes it hard to draw conclusions based on this score. The mutation adequacy score gets overly optimistic when the set of mutants has a lot of redundancy. Moreover, comparing the score for two techniques might be unfair when using the redundant mutants since a technique can score very high while still missing the non-redundant mutants. Work in the direction of identifying redundant mutants is therefore, of great importance to increase the validity in empirical studies on software testing.

One of the more promising approaches to identify redundant mutants is based on a theory of subsumption. Subsumption relations have previously been defined for different software testing criteria. A criterion c_i subsumes another criterion c_j if any test suite that is 100% adequate for c_i is *guaranteed* to also be 100% adequate for c_j. The theory of subsumption

TABLE I
RELATIONAL MUTATION OPERATOR ROR

Original operator	Mutants						
	Mutant M1	Mutant M2	Mutant M3	Mutant M4	Mutant M5	Mutant M6	Mutant M7
$<$	FALSE	$<=$	$!=$	$==$	$>$	TRUE	$>=$
$>$	FALSE	$>=$	$!=$	$==$	$<$	TRUE	$<=$
$<=$	TRUE	$<$	$==$	$!=$	$>=$	FALSE	$>$
$>=$	TRUE	$>$	$==$	$!=$	$<=$	FALSE	$<$
$==$	FALSE	$<=$	$>=$	$<$	$>$	TRUE	$!=$
$!=$	TRUE	$<$	$>$	$<=$	$>=$	FALSE	$==$

is somewhat different when it comes to mutation. Instead of defining subsumption at a criteria level, subsumption is defined as a relation between individual mutants at an operator level. A mutant m_i that is subsumed by another mutant m_j is *provable guaranteed* to be detected by any test that detects m_j. Subsumption relations between mutants have been defined for logic-based mutation operators [4][5]. There is also recent work showing that such relations can be identified for any set of mutants by considering the software under test [6].

II. NON-REDUNDANT ROR MUTANTS

In this work we take a closer look on one of the logic-based mutation operators, the relational mutation operator (*ROR*) and the subsumption theory on ROR mutants. For each instance of any of the six relational operators, the ROR operator originally creates seven mutants. These mutants are shown in Table I. Column 1 in Table I shows the six relational operators and column 2-8 shows the mutants generated for each of these relational operators. Since relational operators are frequently used in software, ROR alone will create a lot of mutants. The majority of these mutants are redundant according to the theory presented by Kaminsky et al. [4][5].

Kaminsky et al. [4][5] address the problem of redundant mutants by proposing fault hierarchies for the *ROR* mutants. A fault hierarchy describes subsumption relations between the seven ROR mutants that are created for the same instance of a relational operator in the original software. A generic fault hierarchy is shown in Figure 1 and this hierarchy can be instantiated for each relational operator from Table I in column 1 using the mutants in the same row. According to the theoretical results [4][5], all the mutants in columns 5-8 (*M4-M7*) in Table I are subsumed by the mutants in columns 2-4 (*M1-M3*). This means that a test suite, which is 100% adequate for the mutants in columns 2-4 is *guaranteed* to be 100% adequate for the mutants in columns 5-8. The mutants in columns 5-8 can therefore be ignored, thus reducing the number of ROR mutants by 57 %.

An arrow between two mutants in Figure 1, $a \rightarrow b$ symbolizes the subsumption relation between the mutants, where the subsumed mutant *b* is redundant. We quote, *"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."* [4].

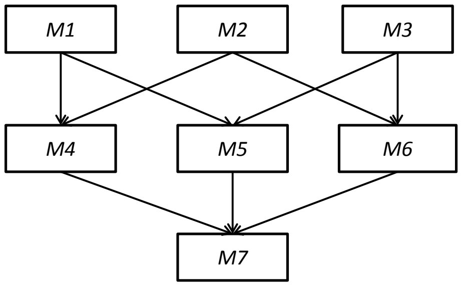

Fig. 1. Generic fault hierarchy for the ROR mutants [4]

$M3 \rightarrow M5$ in Figure 1 should therefore, be interpreted as any test that detects mutant *M3* will also detect mutant *M5*. Moreover, *M7* is also subsumed by *M3* by transitivity. Therefore, it would be sufficient to create the three mutants at the top of the figure, *M1*, *M2* and *M3*. The reason is that any test suite that detects these three top-level mutants would also detect the rest of the mutants. For example, a test that detects that a $>=$ has been mutated to a $==$ is guaranteed to detect that the same $>=$ is mutated to a $<=$.

Moreover, the absence of any arrow between the two top-level mutants *M1* and *M3* should be interpreted as the same test can not detect both *M1* and *M3*. Hence, detection of all three top-level mutants requires at least three different tests.

A. Research Questions

The work of Kaminsky et al. [4][5] is innovative and interesting. The results are well founded when it comes to the arguments for subsumption and the reduced ROR has therefore, been used in several studies [7][8][9]. However, new theories should be empirically evaluated before embraced as valid. The theory is proved using truth assignment and we find this proof solid for an immediate evaluation of a condition. However, the proof is based on the truth assignment of the expression containing the relational operator in isolation without considering what happens afterwards. In fact, there is no discussion on the requirements on fault propagation and it is well known that behaviors can be masked before they propagate to an observable state. The two claims in the quote above are conclusions that, to our knowledge have never been subject to any empirical evaluation. Hence, there are some questions regarding the conclusion validity of the theory. We have therefore, conducted an empirical study to validate this in practice.

The first research question that we should answer concerns the subsumption: It is worth to notice here that there is no discussion regarding any requirements on the oracle used to detect the mutants [4][5]. The general conditions that have to be fulfilled to detect a mutant are the same as for detection of real faults. The execution must reach the fault, execution of the fault must lead to an error state (infection) and this error must propagate to the output [10][11]. In mutation testing, this is referred to as *strong mutation*. The last condition can however, be generalized as the error state must propagate to a state that is observable by the oracle [9][12]. In mutation testing, this is referred to as *weak mutation*. Given an oracle that can detect an error state immediately after execution of the mutated instruction, propagation is not required. Important to note however, is that weak mutation might require some level of propagation [13]. This is sometimes referred to as firm mutation [14]. Basically, weak and strong mutation can be viewed as the two extremes of a scale. For example, the observable state might be at a return statement from a function or component.

It is a reasonable assumption that the same oracle is used for the same test case when testing all seven mutants created for the same instance of a relational operator. This means that the same oracle applies to all mutants in the same fault hierarchy. Hence, for any pair of mutants such that $m \rightarrow m'$, a test *t* and an oracle *o* that detects *m* should detect *m'*. Based on this, we can formulate the research question for the subsumption relation between *m* and *m'*:

R1 Can a test case *t* detecting mutant *m* fail to detect a subsumed mutant *m'* using the same oracle?

The second part of the theory that we want to evaluate, concerns the conclusion that a test cannot detect two mutants if there is no subsumption relation between them according to the fault hierarchy. Hence, detection of all three top-level mutants in a fault hierarchy would require three different test cases. The authors' conclusion is based on the possible truth assignments for the original clause and the mutants, i.e., how these can be true or false simultaneously [4][5]. There is however, no discussion on what might happen after execution of the mutated expression. Our second research question therefore, concerns each pair of top-level mutants *m* and *m'* such that ROR is applied to the same occurrence of a relational operator:

R2 Can a single test case *t* detect more than one of the top-level mutants?

B. Related Work

There is an abundance of literature describing different approaches to reduce the set of mutants, which results in faster execution time [15][16]. Selective mutation techniques use a subset of the mutation operators[17]. It is shown that 60% reduction on the number of mutants can still give a mutation score with 99% using the 6-selective technique and it is also shown that five mutation operators (ABS, AOR, LCR, ROR and UOI) are sufficient for both extended branch coverage and 99% mutation coverage [18][3][19]. Such selective techniques do however, not target the redundant mutants. Instead they remove some of the mutation operators.

Mutation sampling is another technique to reduce the set of generated mutants. Mutation sampling uses only a given percent of all the mutants. The selection can be done randomly or using e.g., a Bayes sequential procedure to perform tests to determine the probability that the program is sufficiently tested[20][21][22]. The sampling techniques have a similar problem as the selective techniques, we do not have any guarantee for the quality of the resulting test suite since not all non-redundant mutants might be included. As opposed to the work on reducing the set of mutants, our work focuses on the evaluation of a new theory on redundant ROR mutants.

There is some work describing empirical studies on redundant mutants by Just, Kapfhammer, and Schweiggert [7][8][9][23]. The topic for their studies is efficient mutation testing and they have used the reduced set of mutants to study its impact on effectiveness and efficiency. However, their studies are based on the assumption that the reduced set of mutants is sufficient and they do not investigate the validity of that assumption. Indeed, they list the possibility that this assumption is wrong as a threat to validity. Our study focuses on the assumption itself.

Kurtz, Ammann, Delamaro, Offutt and Deng [6] take the idea of subsumption graphs one step further in order to identify subsumption relations between all mutants and not just the logic-based ones. This can only be done by taking the software under test into consideration. The authors distinguish between true subsumption, dynamic subsumption and static subsumption. The authors conclude that dynamic subsumption graphs can be overly optimistic regarding the subsumption relations while static subsumption graphs can be overly pessimistic. Ammann, Delamaro and Offutt [24] used dynamic subsumption to establish minimal set of mutants. They used the Siemens test suite [25] to show the impact of their approach. The focus in these papers is how to identify subsumption relations for a particular software under test while our work investigate the assumably true subsumption relations between ROR mutants described by the fault hierarchies.

III. Study Design

This section presents the design of our study and the procedure we followed to investigate the hypotheses. The overall procedure is shown in Figure 2. We selected five small programs for our study. The five programs are Cal, CountNatural, Relation, TestPat and TriTyp. All five are toy programs commonly used for studies. The reason for using such small programs is mainly to be able to verify our results and perform part of the final analysis manually. The programs were selected to represent a variation with respect to their code structure. Table II shows some characteristics for the five programs. As you can see from Table II, the code structure differs between the programs.

We used MuJava to create a complete set of ROR mutants. Stillborn and equivalent mutants were removed from the set

TABLE II
OVERVIEW OF SELECTED SOFTWARE

Software	Relational ops	Predicates	Max clauses	Number of loops	Nesting
Cal	5	3	3	1	No
CountNatural	2	2	1	1	Loop including if
Relation	6	0	0	0	No
TestPat	7	6	2	2	Loop including if including loop including if
TriTyp	17	10	3	0	If including if

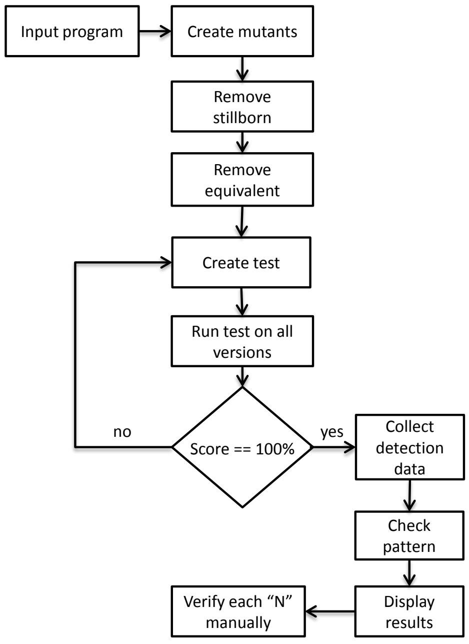

Fig. 2. The procedure used in this study

TABLE III
OVERVIEW OF MUTANTS GENERATED FOR DIFFERENT OPERATORS

Original	Number of mutants
<	26
>	35
<=	55
>=	14
==	78
!=	28

since stillborn mutants do not compile and equivalent mutants cannot be detected by any test. We then iteratively created tests until we had a mutation adequacy score of 100% for all programs. We developed a tool in order to gather necessary data and to search the data for counter examples to the theory and thereby get answers to our research questions.

We ran our test suite against the original versions and all mutants. For each test and each instance of a relational operator in the original software, we gathered information of the status of the mutants as illustrated in Table IV. We used strong mutation so that the output given by a mutant when running a test was compared to the output given by the original when running it with the same test. However, all our programs except Relation includes a single function and nothing more. Hence, strong mutation is in this case very similar to a type of weak (or firm) mutation requiring propagation to a return statement at function level.

TABLE IV
OVERVIEW OF DATA COLLECTED FOR EACH TEST

OrigId	M1	M2	M3	M4	M5	M6	M7
Status	1	0	0	1	1	0	1
OrigId	M1	M2	M3	M4	M5	M6	M7
Status	1	0	1	1	1	1	1
				:			
				:			
OrigId	M1	M2	M3	M4	M5	M6	M7
Status	1	0	0	1	0	0	0

TABLE V
TARGET PATTERNS FOR RESEARCH QUESTION R1

M1	M2	M3	M4	M5	M6	M7
1	any	any	0	any	any	any
1	any	any	any	0	any	any
1	any	any	any	any	any	0
any	1	any	0	any	any	any
any	1	any	any	any	0	any
any	1	any	any	any	any	0
any	any	1	any	0	any	any
any	any	1	any	any	0	any
any	any	1	any	any	any	0
any	any	any	1	any	any	0
any	any	any	any	1	any	0
any	any	any	any	any	1	0

A 0 in Table IV indicates that the mutant is undetected by the test and a 1 indicates that the mutant is detected. This generated pattern were checked against patterns contradicting the theory. Table V and VI illustrates the patterns we looked for. In case the tool found one of these patterns, we manually verified that the result was indeed correct.

TABLE VI
OVERVIEW OF TARGET PATTERNS FOR RESEARCH QUESTION R2

M1	M2	M3	M4	M5	M6	M7
0	1	1	any	any	any	any
1	0	1	any	any	any	any
1	1	0	any	any	any	any
1	1	1	any	any	any	any

IV. Results and Analysis

We made a total of 4642 checks for research question R1 and a total of 622 checks for R2. As shown in Tables VII and VIII, we made several observations contradicting the theory, thus answering the research questions. Worth to note here is that even though the vast majority of the checks were made for TriTyp, we did not see the patterns we looked for in this program. On the other hand, for CountNatural the patterns were observed quite frequently and we found counter examples to both aspects of the theory. We therefore, discuss our findings further using examples based on the same instance of >= in CountNatural. Consider the code below.

```
Public int countNatural(int[] arr)
{
  int count=0;
  for (int i = 0; i < arr.length; i++)
  {
    if (arr[i] >= 0)
    {
      count++;
    }
  }
return count;
}
```

There is a for-loop in CountNatural, which iterates through the array *arr* and counts the non-negative numbers in the array. There are two relational operators in this code, one in the loop condition (<) and one in the if statement (>=). We focus on the operator in the if-statement to illustrate our findings in this section.

TABLE VII
OBSERVATIONS FOR RESEARCH QUESTION R1

SUT	Checks	Pattern Observed R1	Pattern Observed Frequency
Cal	170	0	0
CountNatural	72	6	8.3%
Relation	126	0	0
TestPat	232	16	6.9%
TriTyp	4046	0	0

TABLE VIII
OBSERVATIONS FOR RESEARCH QUESTION R2

SUT	Checks	Pattern Observed R2	Pattern Observed Frequency
Cal	25	3	12%
CountNatural	12	3	25%
Relation	18	0	0%
TestPat	40	0	0%
TriTyp	527	0	0%

A. Subsumption Relation Results

We did find evidence that a test, which detects a top-level mutant may fail to detect one or more of the mutants it subsumes according to the fault hierarchy. In all cases, we used an oracle that observed the state after the return statement. The manual analysis showed that in all cases we identified, there was an intermediate error state, which was corrected before the return statement was reached. Hence, a form of weak mutation that do not require propagation would have detected the subsumed mutants. This result is expected since Kaminsky et al. [4] provide a solid proof for immediate evaluation of the truth assignments.

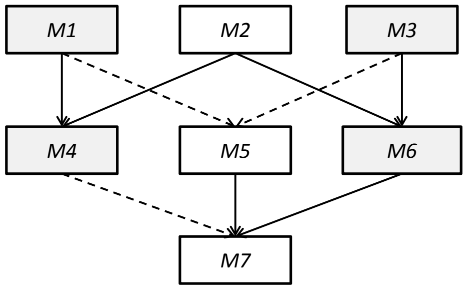
$$\begin{tikzcd}
\boxed{M1} \arrow[d] \arrow[dr, dashed] & \boxed{M2} \arrow[dl] \arrow[dr] & \boxed{M3} \arrow[d] \arrow[dl, dashed] \\
\boxed{M4} \arrow[dr, dashed] & \boxed{M5} \arrow[d] & \boxed{M6} \arrow[dl] \\
 & \boxed{M7} &
\end{tikzcd}$$

Fig. 3. Broken subsumption relations for test *countNatural([1,-1])*

We ran the test *countNatural([1,-1])* and Figure 3 shows the broken fault hierarchy for this test. The Grey mutants are detected and the dotted lines show subsumption relations that do not hold for this test using strong mutation. The detailed results are shown in Table IX. The state of variable *count* as the execution reaches the return statement differs between the different versions of the program. The value of *count* at the return statement is shown for each version in Table IX. In the original program, the value of *count* is 1 as the for loop ends and *count* is returned. Hence, any mutant where $count \neq 1$ at this point of execution is detected by the used oracle.

The original version counts the natural numbers in *arr*. *M1* counts all numbers while *M3* only counts the zeros. Hence both *M1* and *M3* are detected. Given that *M1* and *M3* are detected, *M5* and *M7* should also be detected according to the fault hierarchy for <= shown in Figure 1 and Table I. This is however, not the case and the only reason for why this test does not detect them is that the number of positive and negative numbers in *arr* are the same and that there is no zero in *arr*. *count* is incremented at the wrong points but the right amount of times, which means that the value of variable *count* is correct as the loop ends in *M5* and *M7*.

It is worth to note here that there is an intermediate error state immediately after the incrementation *count++*. This example shows that the fact that an error state propagates to a return statement when running a test on a top-level mutant *m* does not necessarily means that an error state will propagate to the return statement when running the same test on its subsumed mutant *m'*. The error can be masked and disappear

TABLE IX
RESULTS ON TEST *countNatural([1,-1])*

Software version	Code instruction	End state of *count*	Mutant detected
Original	if (*arr[i]* $>=$ 0)	*count*==1	N/A
Mutant M1	if (*true*)	*count*==2	Detected
Mutant M2	if (*arr[i]* $>$ 0)	*count*==1	Not detected
Mutant M3	if (*arr[i]* $==$ 0)	*count*==0	Detected
Mutant M4	if (*arr[i]* $!=$ 0)	*count*==2	Detected
Mutant M5	if (*arr[i]* $<=$ 0)	*count*==1	Not detected
Mutant M6	if (*false*)	*count*==0	Detected
Mutant M7	if (*arr[i]* $<$ 0)	*count*==1	Not detected

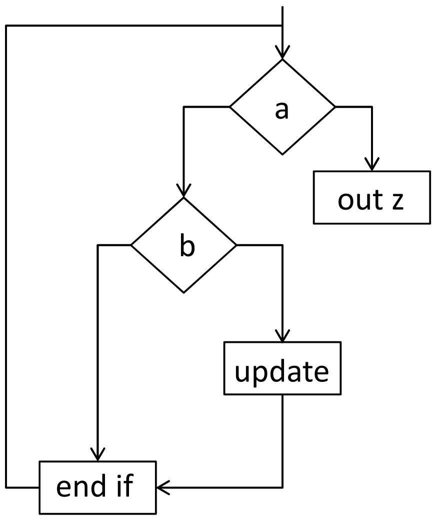

Fig. 4. Iteration that can break the subsumption relations

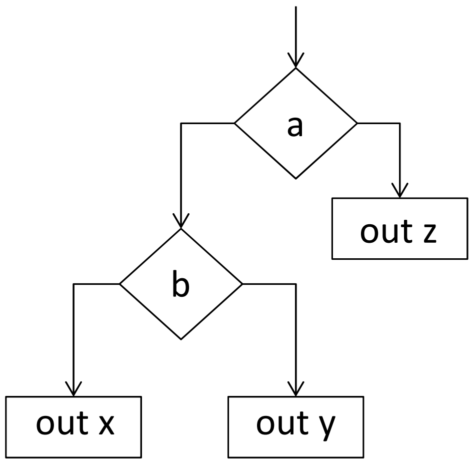

Fig. 5. A nested if-statement do not break the subsumption relations

before it reaches an observable state. The example clearly answers research question R1, it is possible for a test that detects a subsuming mutant *m* to fail to detect a subsumed mutant *m'*. It is not sufficient to use the same oracle. Weak mutation is required and the oracle must observe the state immediately after the modified instruction.

1) Discussion on Propagation Behavior: We noted that all our observations of the pattern we looked for concerned mutants within loops. Iteration, where the mutated instruction is visited more than once seems to be a necessary condition for the observed behavior. We will thus, elaborate a bit further on this. Consider the loop in figure 4. A mutated clause *c* is part of the predicate *b*. Hence, this mutant is visited several times and the effect of an update made during one iteration might be lost during another iteration. This is what we observed in our study.

Now consider the graph in Figure 5. Assume that clause *c* in predicate *a* is subject for mutation and can hence have a different value assignment in the different versions. In order to get a behavior where a top-level mutant is detected, execution of clause *c* must give different truth assignments for the top-level mutant and the original. Moreover, in order for this to make a difference so that the behavior propagates, *c* must be *active*. *c* is active when changing the value of *c* also changes the value of *a* [26]. In this case, the top-level mutant and the original will follow different branches as *a* is evaluated. Moreover, the state up to the point where *c* is evaluated is identical between the versions since only *c* is changed and the test is the same.

If the top-level mutant takes another branch than the original, then so will its subsumed mutants. For example, consider the original $x >= y$, the mutant *M1* (true) and its subsumed mutants *M4* (!=), *M5* (<=) and *M7* (<). *M1* and the original will only follow different branches if $x < y$ in which case the original is false. If $x < y$, then *M4*, *M5* and *M7* will evaluate to true and hence, follow the same branch as *M1*. The same behavior can be seen for each row in Table I. This is the basis for the conclusion drawn by Kaminsky et al. [4] in the quote in Section II.

In order to detect the top-level mutant but not its subsumed mutant, it is necessary to distinguish these two by a second predicate *b*. Note that the mutated clause *c* in predicate *a* is the only difference between the two mutants. Hence, the state of the two mutants are identical before execution of predicate *a* given that the software is predictable. The evaluation of clause *c* then distinguishes the mutants from the original but it does not distinguish the two mutants from each other. Hence, *c* is either true or false in both mutants. The two mutants therefore, still share their states after the execution of decision *a*. Any

TABLE X
RESULTS ON TEST *countNatural([1,1,1,-1,-1,0])*

Software version	Code instruction	End state of *count*	Mutant detected
Original	if (*arr[i]* $>=$ 0)	*count*==4	N/A
Mutant M1	if (*true*)	*count*==6	Detected
Mutant M2	if (*arr[i]* $>$ 0)	*count*==3	Detected
Mutant M3	if (*arr[i]* $==$ 0)	*count*==1	Detected
Mutant M4	if (*arr[i]* $!=$ 0)	*count*==5	Detected
Mutant M5	if (*arr[i]* $<=$ 0)	*count*==3	Detected
Mutant M6	if (*false*)	*count*==0	Detected
Mutant M7	if (*arr[i]* $<$ 0)	*count*==2	Detected

update made after predicate *a* would therefore, affect both mutants equally and they will follow the same branch. This is why we do not see any counter examples to the theory when executing TriTyp or Relational.

Important to note here is that there are exceptions from the behavior described above although we consider them out of scope for the study. If there is unpredictability introduced by e.g., a random value assignment or a race condition, the state of the two mutants might differ as the execution reaches predicate *a* or *b* in Figure 5. If the state differs as *a* is reached, it is possible that the subsumed mutant and the original follow one branch and the top-level mutant another. If the state differs as *b* is reached, it is possible that the error caused by the mutated clause propagates in one mutant and not the other. In fact, if the software is unpredictable there is no guarantee that the test will cause propagation in the same mutant twice if repeated. Repeatability is a well known issue when testing concurrent software. On the other hand, predictable software (or means to control the execution) is assumed in most of the literature on software testing and hence, a very reasonable assumption in this context.

B. Detecting Top-Level Mutants

We did find several examples where more than one top-level mutant were detected by the same test. Consider the example above again. As you can see in Table IX and Figure 3, *M1* and *M3* are both detected by the given test. This should not be possible according to the fault hierarchy in Figure 1 since there is no arrow between *M1* and *M3*. *M1* and *M3* have no subsumption relation between them, not even by transitivity.

Moreover, we did find an example proving that it is actually possible to kill **all** the ROR mutants in the same fault hierarchy by a single test case, (see Figure 6). Table X displays the result of running the test with *countNatural([1,1,1,-1,-1,0])* on the code described in Section IV-A. As you can see in Table X, this test detects all ROR mutants for operator $>=$ in the if statement in countNatural. Hence, research question R2 is also answered. It is indeed possible for a single test to detect more than one of the top-level mutants.

The fact that several top-level mutants can be detected by the same test has nothing to do with strong or weak mutation. A mutant that is detected using strong mutation would have been detected using weak mutation as well since there has to be an error first to give a failure. The behavior has to do with the fact that the mutated instruction is visited more than once. Given a certain point in execution, if a top-level mutant and the original follow different branches, the other top-level mutants will follow the same branch as the original. This is what the results presented by Kaminsky et al. [4] shows. This does however, not mean that the other top-level mutants will follow the same branch as the original the next time they visit the mutated instruction which resides within a loop. The three top-level mutants can be distinguished from the original in different iterations of the loop. Kaminsky et al. [4] draw their conclusions too far when they claim that different tests are needed.

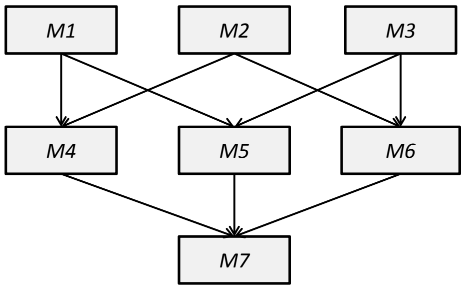
$$\begin{tikzcd}
M1 \arrow[d] \arrow[dr] & M2 \arrow[dl] \arrow[dr] & M3 \arrow[dl] \arrow[d] \\
M4 \arrow[dr] & M5 \arrow[d] & M6 \arrow[dl] \\
 & M7 &
\end{tikzcd}$$

Fig. 6. Three top-level mutants detected by test *countNatural([1,1,1,-1,-1,0])*

C. A Killing Example

Finally, let us look at a third example, which shows that it is actually possible to run a test that detects all the top-level mutants and still misses one of the subsumed. Consider the test *countNatural([1,-1,0,-1,1])*. The result is shown in Table XI and in Figure 7. As you can see, *M1*, *M2*, and *M3* are all detected by the test. *M5* is however, not detected. Again, to detect *M5* by this test, weak mutation is required.

V. CONCLUSIONS

The most important conclusion here is that the theory on redundant ROR mutants does not hold for strong mutation and that a single test can detect more than one top-level mutant independent of strong or weak mutation.

The theoretical results on redundant mutants and subsumption presented by Kaminsky et al. [4] are very interesting and the approach of using fault hierarchies to identify redundant mutants is innovative and promising. Together with resent

TABLE XI
RESULTS ON TEST *arr==[1,-1, 0, -1, 1]*

Software version	Code instruction	End state of *count*	Mutant detected
Original	if (*arr[i]* $>=$ 0)	*count*==3	N/A
Mutant M1	if (*true*)	*count*==5	Detected
Mutant M2	if (*arr[i]* $>$ 0)	*count*==2	Detected
Mutant M3	if (*arr[i]* $==$ 0)	*count*==1	Detected
Mutant M4	if (*arr[i]* $!=$ 0)	*count*==4	Detected
Mutant M5	if (*arr[i]* $<=$ 0)	*count*==3	Not detected
Mutant M6	if (*false*)	*count*==0	Detected
Mutant M7	if (*arr[i]* $<$ 0)	*count*==2	Detected

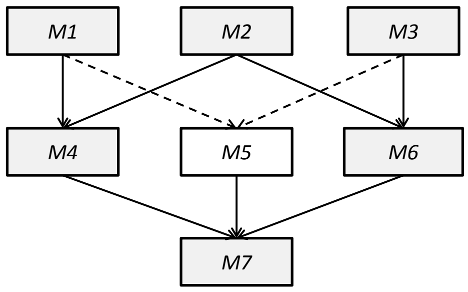

Fig. 7. Broken subsumption relations for test *countNatural([1,-1,0,-1,1])*

approaches for dynamic subsumption [6][24], these ideas have a potential to reduce the effort of mutation analysis tremendously without having the negative side-effect or decreasing the quality of the test suite. There are however, some problems with the conclusions concerning the subsumption theory on ROR mutants.

We have shown that Kaminsky et al. [4] draw their conclusion on the guarantee for detection of a subsumed mutant too far as they do not consider the question of propagation. Their conclusion with respect to the subsumption between ROR mutants does not hold for strong mutation. Moreover, there are different types of weak mutation [13] and in order for the subsumption defined by the fault hierarchy to be guaranteed, the oracle must observe the state immediately after the mutated instruction is executed. Variants of weak (or firm) mutation that require propagation is not sufficient.

We have also shown that a part of the theory is incorrect. According to Kaminsky et al. [4], it should not be possible to detect two top-level mutants in the same fault hierarchy by the same test. Their conclusion is based on the possible truth assignments for the original clause and the mutants, i.e., how these can be true or false simultaneously. However, the authors do not consider the fact that a mutated clause can be visited several times and that the truth assignment to this clause might differ between iterations. By counter examples, we have shown that this part of the theory is not correct. In fact, we have shown that it is possible to detect all three top-level mutants with a single test.

A. Discussion

So what do our results mean in practice? It is well known that redundant mutants can give an overestimation of the mutation score [23] but what if we use the reduced set of ROR mutants together with a strong mutation approach? Will the use of a reduced set affect the test quality or the research results? We believe it might. We have observed that it is possible to create a test suite that is 100% adequate for the reduced set of mutants while not being 100% adequate for the complete set. This shows that there are faults, which might slip through because of the reduction of the set of mutants. It is therefore, important that testers are aware of the fact that the reduced set of mutants *requires* weak mutation in order to guarantee maintained test quality.

Including redundant mutants in a study to evaluate the quality of a test suite might give an overestimation of the results. Reducing the set of mutants by removing mutants that are not redundant might on the other hand, give an underestimation of the results. Hence, when researchers report mutation scores from empirical studies, it should be clarified whether a reduced set is used and whether the score is based on strong or weak mutation.

It is worth to notice here that the dynamic approach taken by [6] to identify subsumption relations by taking the software under test into consideration does not suffer from the problem that we have identified. For such approach, it does not matter whether strong or weak mutation is used since the subsumption relations are identified based on the actual observations made during execution. The drawback with the dynamic approach is of course that the mutants need to be analyzed before the set can be reduced.

For two of the programs, we observed that the error state of the subsumed mutant did not propagate outside the loop in 6.9 and 8.3 % cases. We used strong mutation but since these toy programs only have one function, this result is probably very similar to what we would see in a larger program with an oracle observing the state after the loop. It is likely that the problem of no propagation increases the further the execution proceeds before the oracle observes the state. On the other hand, 6.9 and 8.3 % indicates that the potential loss in test quality by using the three top level mutants instead of all seven still might be acceptable in practice. This number is calculated for individual tests and not the combined effect of the entire test suite. The probability that mutants for which the subsumption do not hold, will be discovered by chance by other tests in a test suite might actually be very high. This is something we have not investigated. Also, a 100%

mutation score might in reality not be achieved due to other problems. For example, it might be hard to distinguish between equivalent mutants and mutants that are just hard to detect (stubborn)[11]. Seen in this light, the gain of reducing the number of mutants by 57% might be much bigger than the potential loss of test quality due to the masking effect we observed. More studies are however, needed to establish any confidence for the extent of the problem. The goal of our study was merely to find counter examples to the theory, not to establish the frequency of the fault slip-through.

B. Threats to Validity

We have shown counter examples to the theory on subsuming ROR mutants and the very existence of these counter examples suggests that the validity for our results is very high when it comes to providing answers to the two research questions.

We built the tool, which was used to find the counter examples ourselves. We cannot guarantee that the implementation of this tool is 100% correct. Most likely, it is not. We therefore, cannot guarantee that we found all the counter examples for our tests and programs. However, whenever the tool found a counter example, we manually verified that the result was correct. All found counter examples are valid and several are presented in this work. Moreover, we only need one counter example to provide sufficient proof that such examples exist.

Only three of the programs in our study had the type of structure that can prevent error propagation (a loop allowing for re-evaluation of the mutated decision) and we only observed the behavior in two of the programs for each research question. The fact that we only found counter examples in two of five programs for each question shows the risk of using small toy examples in this type of study. This type of structure is very common in software and it is likely that we should have made a lot more observations if we used more realistic sized software. Hence, the confidence for how often a test might miss a subsumed mutant in spite of detecting the subsuming mutant is very low. On the other hand, this was not in fact our goal. We just wanted to show the existence of such behavior.

VI. Future Work

The results we have raises a couple of questions regarding how the reduced set of mutants actually affects test quality. Is 6.9-8.3 % a representative number for real-world software? What would the number be if we used even stronger mutation, e.g., checking component or program output. A larger sample of more complex software in terms of nesting levels and tested using strong and weak mutation of different type could give more exact answers to these questions.

We know from our results that a test suite that is 100% adequate for the reduced set of mutants might miss some of the mutants in the complete set. What we do not know is the probability that this happen. The observations that we have made are based on single tests. The fact that a test that should have detected a specific mutant failed to do so does not mean that the mutant cannot be detected by the test suite. In fact, our test suites were generated to be 100% adequate for the complete set of ROR mutants. Hence, it would be interesting to generate test suites for the reduced set of ROR mutants and see what mutation score they get for the complete set. Again, this should be done for a larger sample of more complex software to generate any trustworthy results.

VII. Acknowledgement

This work was funded by The Knowledge Foundation (KKS) through the project 20130085 Testing of Critical System Characteristics (TOCSYC).

References

[1] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[2] N. Li, U. Praphamontripong, and J. Offutt, "An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage," in *Proceedings of the 9th IEEE internationa conference on software testing, verification and validation workshops (ICSTW)*, 2009, pp. 220–229.

[3] J. Offutt, A. Lee, G. Rothermel, R. Untch, and C. Zapf, "An experimental determination of sufficient mutation operators," *ACM Transactions on software engineering methodology*, vol. 5, no. 2, pp. 99–118, april 1996.

[4] G. Kaminski, P. Amman, and J. Offutt, "Improving logic-base testing," *Systems and software*, vol. 86, no. 8, pp. 2002–2012, 2013.

[5] G. Kaminski, P. Ammann, and J. Offutt, "Better predicate testing," in *Proceedings of the 6th International Workshop on Automation of Software Test*. ACM, 2011, pp. 57–63.

[6] B. Kurtz, P. Ammann, M. E. Delamaro, J. Offutt, and L. Deng, "Mutant subsumption graphs," in *IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2014, pp. 176–185.

[7] R. Just, G. M. Kapfhammer, and F. Schweiggert, "Do redundant mutants affect the effectiveness and efficiency of mutation analysis," in *in Proceedings of the Fifth IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2012, pp. 720–725.

[8] R. Just and F. Schweiggert, "Higher accuracy and lower run time: efficient mutation analysis using non-redundant mutation operators," *Software Testing, Verification and Reliability*, vol. 24, 2014.

[9] R. Just, G. M. Kapfhammer, and F. Schweiggert, "Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis," in *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*. IEEE, 2012, pp. 11–20.

[10] J. M. Voas, "Pie: A dynamic failure-based technique," *Software Engineering, IEEE Transactions on*, vol. 18, no. 8, pp. 717–727, 1992.

[11] X. Yao, M. Harman, and Y. Jia, "A study of equivalent and stubborn mutation operators using human analysis of equivalence," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 919–930.

[12] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *Software Engineering, IEEE Transactions on*, vol. 38, no. 2, pp. 278–292, 2012.

[13] A. J. Offutt and S. D. Lee, "An empirical evaluation of weak mutation," *IEEE Transactions on Software Engineering*, vol. 20, no. 5, pp. 337–344, 1994.

[14] M. Woodward and K. Halewood, "From weak to strong, dead or alive? an analysis of some mutation testing issues," in *Software Testing, Verification, and Analysis, 1988., Proceedings of the Second Workshop on*. IEEE, 1988, pp. 152–158.

[15] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[16] A. J. Offutt and R. H. Untch, "Mutation 2000: Uniting the orthogonal," in *Mutation testing for the new century*. Springer, 2001, pp. 34–44.

[17] E. S. Mresa and L. Bottaci, "Efficiency of mutation operators and selective mutation strategies: An empirical study," *Software Testing Verification and Reliability*, vol. 9, no. 4, pp. 205–232, 1999.

[18] A. J. Offutt, G. Rothermel, and C. Zapf, “An experimental evaluation of selective mutation,” in *Proceedings of the 15th International Conference on Software Engineering (ICSE93)*, 1993, pp. 100–107.

[19] W. E. Wong, J. C. Maldonado, M. E. Delamaro, and A. P. Mathur, “Constrained mutation in c programs,” in *Proceedings of the 8th Brazilian Symposium on Software Engineering*, 1994, pp. 439–452.

[20] W. E. Wong and A. P. Mathur, “Reducing the cost of mutation testing: An empirical study,” *Journal of Systems and Software*, vol. 31, no. 3, pp. 185–196, 1995.

[21] R. A. DeMillo, D. S. Guindi, W. M. McCracken, A. J. Offutt, and K. N. King, “An extended overview of the mothra software testing environment,” in *”IEEE” Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, 1988, pp. 142–151.

[22] M. Sahinoglu and E. H. Spafford, “Sequential statistical procedures for approving test sets using mutation-based software testing,” in *Proceedings of the IFIP Conference on Approving Software Products (ASP 90)*, 1990.

[23] R. Just, “The major mutation framework: Efficient and scalable mutation analysis for java,” in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 433–436.

[24] P. Ammann, M. E. Delamaro, and J. Offutt, “Establishing theoretical minimal sets of mutants,” in *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*. IEEE, 2014, pp. 21–30.

[25] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, “Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria,” in *Proceedings of the 16th international conference on Software engineering*. IEEE Computer Society Press, 1994, pp. 191–200.

[26] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2008.