IJDCST @Nov-Dec, Issue- V-3, I-1, SW-76
ISSN-2320-7884 (Online)
ISSN-2321-0257 (Print)

Metamorphic Relations for Program Testing, Debugging using an Enhancement Method

M.Vijaya Kumar[1], D.HariKrishna[2], Dr. K. Rama Krishnaiah[3]

[1] Dept. of CSE, Nova College of Engineerng & Technology,Vijayawada,AP,India.

[2]Assistant Professor, Nova College of Engineerng & Technology,Vijayawada,AP,India.

[3] Professor, Nova College of Engineerng & Technology,Vijayawada,AP,India.

ABSTRACT: A normally utilized methodology is to actualize distinctive plans at different stages for a powerful program. In this paper, we shows an improved system for project testing, and debugging that uses metamorphic relations(MR).for program accuracy the strategy guarantees that a system fulfills chose program properties(metamorphic relations) for a scope of information spaces. For project testing the strategy itself is a computerized typical testing system that is utilized to test chose program ways for taking care of unobtrusive blames in programming testing, for example, the missing way slips. In proposed framework, we supplant the backtracking calculation with "SYNERGY" algorithm that joins testing and demonstrating to check program properties. At last the strategy likewise upholds programmed debugging through the recognizable proof of stipulations for disappointment creating inputs. The proposed framework is perfect for medium and vast scale applications.

Keywords: Metamorphic, SYNERGY, Testing and Debugging

I. INTRODUCTION

Program accuracy has dependably been a discriminating issue for both analysts and experts. The previous decades have demonstrated that the utilization of formal check (i.e., system demonstrating) to genuine applications has been exceptionally restricted because of the troubles in evidences and computerization. Project testing, hence, remains the most well known means embraced by specialists. By and by, testing has two major impediments. First and foremost, the utilization of experiments can't promise program accuracy on untested inputs. As it were, trying can't demonstrate the unlucky deficiency of shortcomings much of the time. Furthermore, in a few circumstances, it is unimaginable or basically excessively hard to choose whether the system yields on experiments are right. This is known as the prophet issue. As of late new programming testing system, to be specific metamorphic testing, has been proposed to assuage the prophet issue.

Changeable Testing is a computerized testing technique that utilizes expected properties of the target capacities to test projects without human contribution. These properties are called metamorphic relations (MR). The thought of checking the normal properties of target frameworks without being limited to character relations has been utilized in changeable testing and the testing of equality and non-equality of articles.

Utilizing the idea of metamorphic relations, we choose essential properties for the target program. At that point we perform typical executions. This is on account of the yield of a typical execution is more instructive than that of executing a cement info, as an issue data speaks to more than one component of the information area. We will utilize the term typical executions to allude to the executions of chose ways with chose typical inputs, and the term worldwide typical assessment for the executions of all conceivable ways of the project with typical inputs covering the whole include space. For projects that are excessively intricate for worldwide image assessment or imperative solvers, our methodology can at present be connected as an issue testing methodology.

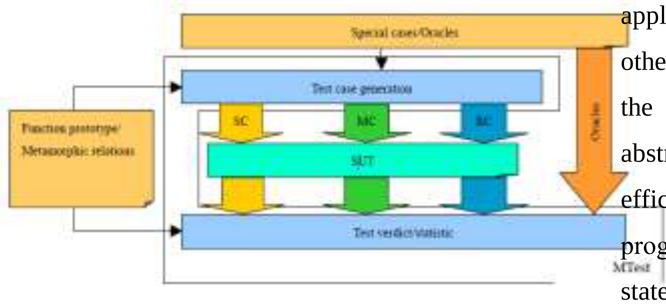


Fig.1: Metamorphic test architecture.

In existing framework, Individual methodologies are utilized for project demonstrating, testing and debugging. There is no Integrated Model for bringing together all the three. Later, an Integrated Method called semi demonstrating is utilized for Program Correctness, Testing, and Debugging utilizing Metamorphic Relations (MR). Semi Proving uses four Mrs that are very not quite the same as each other with a perspective to recognize different flaws, since it is profoundly farfetched for a solitary MR to identify all the deficiencies. For Testing, Semi Proving uses a backtracking algorithm of Pathfinder model checker to cross the typical execution tree as opposed to beginning without any preparation for each typical execution. The backtracking calculation is not a helpful methodology for medium and huge scale applications.

In this paper, we demonstrates an enhanced method for program testing, and debugging that uses metamorphic relations(MR).For program correctness the method ensures that a program satisfies selected program properties(metamorphic relations) for a range of input domains. For testing, we replace the backtracking algorithm with “SYNERGY” algorithm that combines testing and proving to check program properties. In our approach, Synergy, a testing method is combined with Metamorphic Relations the system is ideal for medium and large scale applications. SYNERGY is different from that of other testing methods. It does not attempt to traverse the execution tree; instead, it attempts to cover all abstract states (equivalence classes.) It can be more efficient in constructing proofs of correctness for programs with the “diamond” structure of if-then-else statements. Finally the method also supports automatic debugging through the identification of constraints for failure-causing inputs.

II. Related Work

A strategy has been created by Yorsh et al. to join testing, deliberation, and hypothesis demonstrating. Utilizing this technique, system states gathered from executions of cement experiments are summed up by method for reflections. At that point, a hypothesis prover will check the summed up set of states against a scope basis and against certain security properties. At the point when the check is effective, the wellbeing properties are demonstrated. Yorsh et al's. system "is situated towards discovering an evidence as opposed to recognizing true blunders," and "does not recognize a false mistake and a genuine slip."

Zeller and Hildebrandt proposed a Delta Debugging calculation that changes a disappointment bringing about information into an insignificant structure that can at present fizzle the project. This is carried out by persistently narrowing down the distinction between disappointment creating and non-disappointment bringing about inputs. Zeller further created the Delta Debugging system by looking at "what's going ahead inside the project" amid the execution. He considered the fizzled execution as an issue of system states, and just piece of the variables and values in a percentage of the states are important to the disappointment. He proposed disengaging these important variables and values by ceaselessly narrowing the distinction in system states in fruitful and fizzled executions.

He and Gupta acquainted a methodology with both finding and adjusting defective proclamations in a project under test. The methodology joins thoughts from accuracy demonstrating and programming testing to find an imaginable mistaken proclamation and afterward right it. It expect that a right determination of the system under test is given as far as preconditions and post conditions. It likewise expect that stand out articulation in the project under test is at shortcoming. Utilizing the idea of way based weakest precondition, the thoughts of a theorized system state and a real program state at each point along the disappointment way (execution follow) are characterized. He and Gupta's calculation crosses the disappointment way and contrasts the states at each one point with identify proof for a conceivable broken proclamation. Such "proof" will rise if a predicate speaking to the real program states is less prohibitive than the predicate speaking to the Hypothesized project states. The calculation then creates an alteration to the probable broken explanation. The altered system is then tried utilizing all current experiments.

III INTEGRATED METHOD OVERVIEW

Let p be the project under test, t be the starting effective experiment, R be a MR, and t′ be the catch up experiment produced as indicated by R. For simplicity of presentation and comprehension, let us focus on Mrs that is personality relations. For non-character relations, the examination will be comparative. Thus, it is the connection "$p(t) = p(t')$" that is weighed in MT. Our point is to choose such a MR, to the point that has a higher opportunity to cause $p(t) \neq p(t')$. We propose the accompanying speculation:

For a defective system p and a couple of changeable experiments (t, t′), as a rule the more the execution of p(t′) contrasts from the execution of p(t), the more probable it is that their yields are not equivalent.

We have not expressly characterized the idea of "distinction between two executions". This idea covers all parts of project executions, including the ways navigated, arrangement of the announcements worked out, succession of diverse qualities appointed to variables, and so forth. Taking into account Hypothesis, our MR choice technique is to choose such Mrs that can make the two executions as distinctive as would be prudent. For project p(t), the info t is a tuple including one or more parameters, i.e., $t = (x1, x2, . . . , xn)$, where $n \geq 1$. Normally, diverse xi's ($1 \leq i \leq n$) assume distinctive parts in the execution and, henceforth, they have diverse impact on the general execution stream (i.e., ways executed, variable qualities, cycle times, and so on.) Hence, we propose selecting those Mrs that can change the estimations of the basic parameters as extraordinarily as could be expected under the circumstances. A "discriminating parameter" is such a xi in t, to the point that assumes the most essential part in controlling how the system is to be executed. The catch up experiment t′ hence created will, in this way, constrain an altogether different execution.

We represented how to demonstrate that the project is right regarding a chose changeable connection. In circumstances where the accuracy of a few yields can be chosen, for example, uncommon worth cases, we can extrapolate from the rightness for tried inputs to the rightness for related untested inputs. Let us, for example, test the project Med with a particular typical experiment (x, y, z) such that $x \leq z \leq y$. The yield is z. We can, obviously, effectively check that z is a right yield. Henceforth, the project breezes through this particular test. We can extrapolate that the yields of the project Med are right for all different inputs as takes after: Suppose I = (a, b, c) is any triple of whole numbers. Given i a chance to) (= (a′, b′, c′) be a stage of I such that $a' \leq c' \leq b'$. As per the aftereffect of the typical testing above, Med (a′, b′, c′) = average (a′, b′, c′). The way that (a′, b′, c′) is essentially a change of (a, b, c) intimates that average (a′, b′, c′) = average (a, b, c). Consequently, Med (a′, b′, c′) = average (a, b, c). Then again, it has been demonstrated that Med (a′, b′, c′) = Med (a, b, c). In this manner, Med (a, b, c) = average (a, b, c). Thusly, we have demonstrated that the yields of Med (a, b, c) are correct for any input. In other words, the correctness is extrapolated from tested symbolic inputs to untested symbolic inputs.

In routine programming testing (counting metamorphic testing), solid disappointment creating inputs are distinguished yet not the interrelationships among them. In our methodology, backings debugging by giving express depictions of the interrelationships among changeable disappointment bringing about inputs through MFCC. Contrasted and cement disappointment bringing about inputs, such interrelationships contain more data about the deformities. Contrasted and changeable testing, our system has an alternate preference notwithstanding its backing of debugging: It has a higher flaw identification ability.

"Testing is concerned with issue recognition, while placing and diagnosing flaws fall under the rubric of debugging. "Spotting the imperfections" ought not just be deciphered as the distinguishing proof of flawed explanations in a project. We have actualized confirmation and debugging framework with an alternate centering. Our framework produces demonstrative data on the reason impact affix that prompts a disappointment. We characterize a metamorphic preserving condition (MPC) as an issue under which a system fulfills an endorsed metamorphic connection. When we recognize a MFCC, a relating MPC can likewise be distinguished in the meantime. When the trigger is recognized, the debugger will further stand up in comparison the execution follows, way conditions, etc, and after that report the distinctions as an issue impact fasten that prompts the disappointment.

In our verification system, the source code of any program under test is instrumented using a program instrument or prior to compilation, so that an execution trace can be collected. When a violation of a metamorphic relation occurs, a failure report will be generated in two steps. First, details of the initial and follow-up executions are recorded. Then, diagnostic details are added. When there are a large number of paths to verify, the efficiency of symbolic-execution-based approaches is also a concern. There are, however, algorithms and tools that tackle such tasks more efficiently. The Java PathFinder model checker, for example, uses a backtracking algorithm to traverse the symbolic execution tree instead of starting from scratch for every symbolic execution.

To achieve higher scalability for large software applications, in our approach we replace the backtracking algorithm with “SYNERGY” algorithm that combines testing and proving to check program properties. It unifies the ideas of counter example guided model checking, directed testing , and partition refinement.

SYNERGY is different from that of other testing methods. It does not attempt to traverse the execution tree, instead, it attempts to cover all abstract states (equivalence classes.) It can be more efficient in constructing proofs of correctness for programs with the “diamond” structure of if-then-else statements.

IV OVERVIEW ON SYNERGY

We present a new verification algorithm, called Synergy, which searches simultaneously for bugs and proof, and while doing so, tries to put the information obtained in one search to the best possible use in the other search. The search for bugs is guided by the proof under construction, and the search for proof is guided by the program executions that have already been performed.

Synergy keeps up two information structures. For the under estimated (solid) investigation, Synergy gathers the test runs it executes as an issue F. Every way in the woods F relates to a cement execution of the system. The timberland F is become by performing new tests. When a blunder area is added to F, a true slip has been found. For the overestimated (conceptual) examination, Synergy keeps up a limited, social reflection An of the system. Each one condition of An is an equivalence class of cement system states, and there is a move from unique express an excessively theoretical state b if some solid state in a has a move to some solid state in b. At first, A contains one dynamic state for every system area.

Synergy grows develops the backwoods F by taking a gander at the part An, and it refines A by taking a gander at F. At whatever point there is an (unique) blunder way in A, Synergy picks a lapse way τerr in A which has a prefix τ such that (1) τ compares to a (cement) way in F, and (2) no theoretical state in τerr after the prefix τ is gone by in F. Such a "requested" way τerr dependably exists. Collaboration now tries to add to F another test which takes after the requested way τerr for no less than one move past the prefix τ . We utilize coordinated testing to check if such a "suitable" test exists. In the event that a suitable test exists, then it has a decent risk of hitting the slip if the mistake is in fact reachable along the requested way. Also regardless of the possibility that the suitable test does not hit the blunder, it will show a more extended doable prefix of the requested way. Then again, if a suitable test does not exist, then as opposed to developing the woods F, Synergy refines the part A by evacuating the first conceptual move after the prefix τ along the requested way τ error. At that point Synergy proceeds by picking another requested way, until either F discovers a genuine system lapse or A gives a confirmation of project.

V SYNERGY VERIFICATION

The algorithm Synergy takes as inputs (1) a program P = <Σ, σ, →>, and (2) a property ψ where Σ is a set of states, σ is initial state and → is transaction relation. It can produce three types of results:

1. It may output “fail” together with a test generating an error trace of P to ψ.

2. It may output "pass" together with a finite-indexed partition Σ≈ proving that P cannot reach ψ.
3. It may not terminate.

It maintains two core data structures: (1) a finite forest F of states, where for every state s Є F, either s Є σ or parent (s) Є F is a concrete predecessor of s (that is, parent (s) →s); and (2) a finite-indexed partition Σ≈ of the state space Σ.

At first, F is void, and Σ≈ is the starting allotment with three districts, to be specific, the introductory states σ, the slip states ψ, and all different states. In every emphasis of the fundamental circle, the calculation either stretches the backwoods F to incorporate more reachable states, or refines the segment Σ≈. Theoretical blunder follows are utilized to run experiment era and the non-presence of specific sorts of experiments is utilized to guide allotment refinement. In every cycle of the circle, the calculation first verifies whether it has effectively discovered an experiment to the mistake district. This is checked by searching for a district S such that S ∩ F =ø and S ⊆ ψ . All things considered, the calculation picks a state s ∈ S ∩ F and calls the assistant capacity Testfromwitness to process an experiment (data vector) that creates a blunder follow. Instinctively, Testfromwitness lives up to expectations by progressively finding the guardian until it discovers a foundation of the timberland F. Formally, for a state s ∈ F, the capacity call Testfromwitness(s) gives back where its due succession s0 , s1, . . . , sn such that sn = s, and guardian (si) = si−1 for every one of the 0 < i ≤ n, and guardian (s0) = €. The beginning state s0 gives the craved experiment.

In the event that it is not ready to discover an experiment prompting the slip, the calculation checks if the current segment Σ≈ gives a confirmation that P can't reach ψ. It does this by first building the dynamic project P≈ utilizing the assistant capacity Createabstractprogram . Given a parcel Σ≈, the capacity Createabstractprogram(p,σ≈) gives back where its due project P≈ = <σ≈, σ≈,→≈>. The following step is to call to the helper capacity Getabstracttrace keeping in mind the end goal to scan for a conceptual mistake follow. In the event that there is no unique lapse follow, then Getabstracttrace furnishes a proportional payback follow €. All things considered, the calculation returns "pass" with the current parcel Σ≈. Something else, Getabstracttrace gives back a dynamic follow s0 , s1, . . . , sn such that Sn ⊆ ψ. The next step is to convert this trace into an ordered abstract trace.

The abstract trace s_0 , s_1, . . . , s_n is ordered if the following two conditions hold:

1. There exists a frontier k def = Frontier(s_0 , s_1, . . . , s_n) such that (a) 0 ≤ k ≤ n, and (b) Si ∩F =ø for all k ≤ i ≤ n, and (c) Sj ∩ F = ø for all 0 ≤ j < k.

2. There exists a state s ∈ Sk−1 ∩ F such that Si = Region(parentk−1−i(s)) for all 0 ≤ i < k, where the abstraction function Region maps each state s ∈ Σ to the region S ∈ Σ≈ with s ∈ S.

We note that whenever there is an abstract error trace, then there must exist an ordered abstract error trace. The auxiliary function GetOrderedAbstractTrace converts an arbitrary abstract trace τ into an ordered abstract trace τerr. Intuitively, it works by finding the latest region in the trace that intersects with the forest F, choosing a state in this intersection, and following the parent pointers from the chosen state. The function GetOrderedAbstractTrace returns a pair <τerr, k>, where τerr is an ordered abstract error trace and k = Frontier(τerr).

The algorithm now tries to extend the forest F along the ordered abstract error trace τerr. We define suitable tests in two steps. First we define F-extensions, which are sequences that can be added to F while still maintaining the invariant that F is a forest. A finite sequence s_0 , s_1, . . . , s_n m of states is an F-extension if (1) s0 ∈ σ, and (2) si→si+1 for all 0 ≤ i < m, and (3) there exists k such that (a) 0 ≤ k < m and (b) si ∈ F for all 0 ≤ i < k and (c) sj ∈ F for all k ≤ j ≤ m. Given an abstract trace τerr = S0 , S1, . . . , Sn with k = Frontier(τerr), and the forest F, a sequence of states is suitable if it is (1) an F-extension and (2) follows the abstract trace τerr at least for k steps. Formally, the auxiliary function GenSuitableTest(τerr, F) takes as inputs an ordered abstract trace τerr = S0 , S1, . . . ,Sn and the forest F, and either returns an F-extension t = s0 , s1, . . . , sm such that (a) m ≥ Frontier(τerr) and (b) si ∈ Si for all 0 ≤ i ≤ Frontier(τerr), or returns € if no such suitable sequence exists.

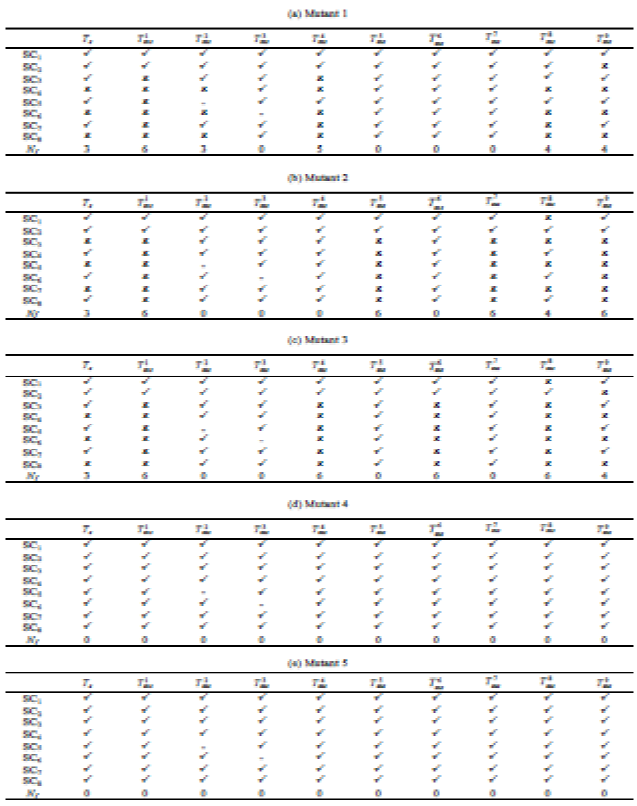
If we succeed in finding such a suitable test case, we simply add it to the forest F, and continue. If no suitable test is found, then we know that there is no concrete program execution corresponding to the abstract trace S0 , S1, . . . , Sk. However, we already have a concrete execution along the prefix S0 , S1, . . . , Sk−1, because Sk−1∩F= ø. Thus, we split the region Sk−1 using the preimage operator Pre(Sk) = {s ∈ Σ | ∃s' ∈ Sk. s → s}, and thus eliminate the spurious (infeasible) abstract error trace from the abstract program. The call to the auxiliary function RefineWithGeneralization has been commented out. This call is needed to help Synergy terminate on certain programs.

VI Experimental Results

This section will illustrate our experimental results on metamorphic testing for sparse matrix multiplication programs, which has been performed automatically with MTest.

Metamorphic testing with special test cases

The special test set consists of 8 test cases, derived from atomic properties mentioned in Section 1.



(a) Mutant 1

	T_s	T^1_{mt}	T^2_{mt}	T^3_{mt}	T^4_{mt}	T^5_{mt}	T^6_{mt}	T^7_{mt}	T^8_{mt}	T^9_{mt}
SC_1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SC_2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗
SC_3	✓	✗	✓	✓	✗	✓	✓	✓	✓	✓
SC_4	✗	✗	✗	✓	✗	✓	✓	✓	✗	✗
SC_5	✓	✗	–	✓	✓	✓	✓	✓	✓	✓
SC_6	✗	✗	✗	–	✗	✓	✓	✓	✗	✗
SC_7	✓	✗	✓	✓	✗	✓	✓	✓	✗	✓
SC_8	✗	✗	✗	✓	✗	✓	✓	✓	✗	✗
N_F	3	6	3	0	5	0	0	0	4	4

(b) Mutant 2

	T_s	T^1_{mt}	T^2_{mt}	T^3_{mt}	T^4_{mt}	T^5_{mt}	T^6_{mt}	T^7_{mt}	T^8_{mt}	T^9_{mt}
SC_1	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓
SC_2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SC_3	✗	✗	✓	✓	✓	✗	✓	✗	✗	✗
SC_4	✓	✗	✓	✓	✓	✗	✓	✗	✓	✗
SC_5	✗	✗	–	✓	✓	✗	✓	✗	✗	✗
SC_6	✓	✗	✓	–	✓	✗	✓	✗	✓	✗
SC_7	✗	✗	✓	✓	✓	✗	✓	✗	✗	✗
SC_8	✓	✗	✓	✓	✓	✗	✓	✗	✓	✗
N_F	3	6	0	0	0	6	0	6	4	6

(c) Mutant 3

	T_s	T^1_{mt}	T^2_{mt}	T^3_{mt}	T^4_{mt}	T^5_{mt}	T^6_{mt}	T^7_{mt}	T^8_{mt}	T^9_{mt}
SC_1	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓
SC_2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗
SC_3	✓	✗	✓	✓	✗	✓	✗	✓	✗	✓
SC_4	✗	✗	✓	✓	✗	✓	✗	✓	✗	✗
SC_5	✓	✗	–	✓	✗	✓	✗	✓	✗	✓
SC_6	✗	✗	✓	–	✗	✓	✗	✓	✗	✗
SC_7	✓	✗	✓	✓	✗	✓	✗	✓	✗	✓
SC_8	✗	✗	✓	✓	✗	✓	✗	✓	✗	✗
N_F	3	6	0	0	6	0	6	0	6	4

(d) Mutant 4

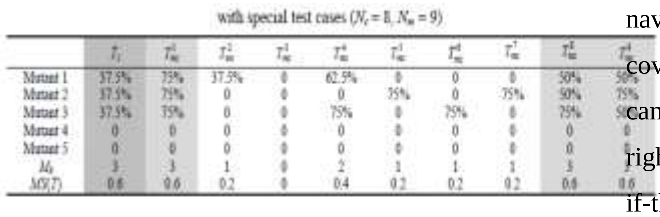
	T_s	T^1_{mt}	T^2_{mt}	T^3_{mt}	T^4_{mt}	T^5_{mt}	T^6_{mt}	T^7_{mt}	T^8_{mt}	T^9_{mt}
SC_1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SC_2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SC_3	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SC_4	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SC_5	✓	✓	–	✓	✓	✓	✓	✓	✓	✓
SC_6	✓	✓	✓	–	✓	✓	✓	✓	✓	✓
SC_7	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SC_8	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
N_F	0	0	0	0	0	0	0	0	0	0

(e) Mutant 5

	T_s	T^1_{mt}	T^2_{mt}	T^3_{mt}	T^4_{mt}	T^5_{mt}	T^6_{mt}	T^7_{mt}	T^8_{mt}	T^9_{mt}
SC_1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SC_2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SC_3	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SC_4	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SC_5	✓	✓	–	✓	✓	✓	✓	✓	✓	✓
SC_6	✓	✓	✓	–	✓	✓	✓	✓	✓	✓
SC_7	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SC_8	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
N_F	0	0	0	0	0	0	0	0	0	0

Table 1 Test verdicts report by special case testing and metamorphic testing with special test cases.

Table 4 reports the mutation score and fault detection ratio of special case testing in Column 2, and metamorphic testing with each metamorphic relation MR_i, in Columns 3 to 11.



with special test cases ($N_c = 8$, $N_m = 9$)

	T_s	T_{mr}^1	T_{mr}^2	T_{mr}^3	T_{mr}^4	T_{mr}^5	T_{mr}^6	T_{mr}^7	T_{mr}^8	T_{mr}^9
Mutant 1	37.5%	75%	37.5%	0	62.5%	0	0	0	50%	50%
Mutant 2	37.5%	75%	0	0	0	75%	0	75%	50%	75%
Mutant 3	37.5%	75%	0	0	75%	0	75%	0	75%	50%
Mutant 4	0	0	0	0	0	0	0	0	0	0
Mutant 5	0	0	0	0	0	0	0	0	0	0
M_k	3	3	1	0	2	1	1	1	3	3
$MS(T)$	0.6	0.6	0.2	0	0.4	0.2	0.2	0.2	0.6	0.6

Table 2 Mutation score and fault detection ratio of special case testing and metamorphic testing.

Based on the above data, the following conclusions can be drawn:

1. Metamorphic testing with a single metamorphic relation, as well as with special test cases, may not outperform special case testing. For sparse matrix multiplication, MR_1, MR_8 and MR_9 seem better than other metamorphic relations, among which MR_3 is the worst one.

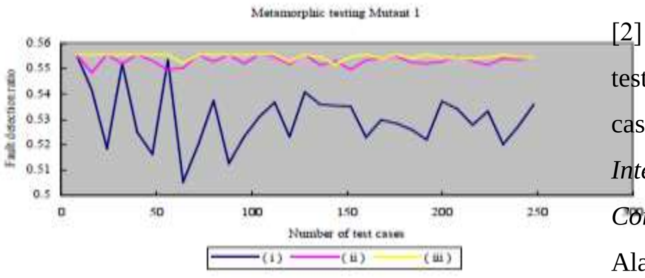


Fig.2: FD(T_{mr}) with increasing number of random test cases and minimum order of a random matrix: (i) D_{min}=2, (ii) D_{min}=6, (iii) D_{min}=18.

CONCLUSION:

We have presented a integrated system for demonstrating, testing, and debugging. Firstly, it demonstrates that the system fulfills chose program properties (that is, changeable relations) all through the whole include area or for a subset of it. For testing, we supplant the backtracking calculation with "Cooperative energy" calculation that consolidates testing and demonstrating to check program properties. In our methodology, Synergy, a testing system is consolidated with Metamorphic Relations the framework is perfect for medium and extensive scale applications. Collaboration is not the same as that of other testing systems. It doesn't endeavor to navigate the execution tree; rather, it endeavors to cover all theoretical states (proportionality classes.) It can be more proficient in developing verifications of rightness for projects with the "diamond" structure of if-then-else explanations. At long last the strategy likewise upholds programmed debugging through the ID of imperatives for disappointment creating inputs. It is essentially intriguing, in light of the fact that it consolidates the capacity of different instruments to handle an extensive number of project ways utilizing a little number of dynamic states and dodge unnecessary refinements through cement execution.

REFFERENCES:

[1] Java PathFinder Home Page. http://javapathfinder. sourceforge.net.

[2] T.Y. Chen, J. Feng, and T.H. Tse, "Metamorphic testing of programs on partial differential equations: a case study," *Proceedings of the 26th Annual International Computer Software and Applications Conference* (*COMP- SAC 2002*), pp. 327–333. Los Alamitos, CA: IEEE Computer Society Press, 2002.

[3] H. Agrawal, J.R. Horgan, S. London, and W. E. Wong, "Fault localization using execution slices and dataflow tests," *Proceedings of the 6th International Symposium on Software Reliability Engineering* (*ISSRE '95*), pp. 143– 151. Los Alamitos, CA: IEEE Computer Society Press, 1995.

[4] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M.D. Ernst, "Finding bugs in dynamic web applications," *Proceedings of the 2008 ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA 2008*), pp. 261–272. New York, NY: ACM Press, 2008.

[5] L. Baresi and M. Young, "Test oracles," Technical Report
CIS-TR01-02, Department of Computer and Information Science, University of Oregon, Eugene, OR, 2001.

[6] T. Ball and S.K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Model Checking of Software* (SPIN), LNCS 2057, pp. 103–122. Springer, 2001.

[7] E.M. Clarke, O. Grumberg, D. Peled. *Model Checking*. MIT Press, 1999.

[8] P. Godefroid, N. Klarlund, K. Sen. Dart: Directed automated random testing. In *Programming Language Design and Implementation*, pp. 213–223. ACM, 2005.

[9] M. Blum, M. Luby, and R. Rubinfeld, "Selftesting / correcting with applications to numerical problems," *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing* (*STOC '90*), pp. 73–83. New York, NY: ACM Press, 1990. Also *Journal of Computer and System Sciences*, vol. 47, no. 3, pp. 549–595, 1993.

[10] B. Burgstaller, B. Scholz, and J. Blieberger, *Symbolic Analysis of Imperative Programming Languages*, Lecture Notes in Computer Science, vol. 4228, pp. 172–194. Berlin, Germany: Springer, 2006.

About Authors:

I am M.Vijaya Kumar pursuing my Mtech from Nova college of Engineering & Technology, My interest are research in data mining & Software Engineering.

Mr. Hari Krishna.Deevi is a qualified persion Holding M.Sc(CSE) & M.Tech Degree in CSE from Acharya Nagarjuna university, He is an Outstanding Administrator & Coordinator. He is working as an Assistant Professor in NOVA College of Engineering Technology .He guided students in doing IBM projects at NOVA ENGINEERING College. Who has Published 10 research Papers in various international Journals and workshops with his incredible work to gain the knowledge for feature errands.

Dr. K. Rama Krishnaiah is a highly qualified person, an efficient and eminent academician. He is an outstanding administrator; a prolific researcher published 33 research papers in various International Journals and a forward looking educationist. He worked in prestigious K L University for 11.5 years and he contributed his service for NBA accreditation in May 2004, Aug 2007 with 'record rating', ISO 9001:2000 in 2004, Autonomous status in 2006, NAAC accreditation of UGC in 2008 and University status in 2009. Later on he worked as Principal at Nova College of Engineering and Technology, Vijayawada for a period of 3.5Yrs. He took charge as the Principal, NVR College of Engineering and Technology, Tenali in May 2014.