International Conference on Computational Science and Information Management (ICoCSIM)
Vol. 1 (2012) 239-250

P2R – A Pairwise Testing Strategy Supporting Execution Resumption

Mohammed H.S. Helal[a] and Kamal Z. Zamli[b]*

[a]School of Electrical and Elecronics, Universiti Sains Malaysia, Engineering Campus, 14300 Nibong Tebal, Penang, Malaysia
[b]Faculty of Computer Systems and Software Engineering, Universiti Malaysia Pahang, Lebuh Tun Razak, 26300 Kuantan, Pahang, Malaysia

Abstract

Pairwise strategies have been proven to be useful to systematically sample test data for interaction testing consideration. However, generating pairwise test suite can be a painstakingly difficult endeavor especially involving large configurations. Here, the generation process can take hours and even days to complete owing to the need to sample from large numbers of input parameters and values. In the case of interruption (e.g. due to power failure or hardware malfunctions), there is often a need to restart the whole generation process again from the beginning. If the pairwise test suite is significantly large and the overall progress is near completion, restarting means a lot of wasted efforts.In order to address the aforementioned issues, we have developed a new pairwise strategy, called P2R, which can generate sufficiently competitive pairwise test suite. Unlike existing pairwise strategies, P2R also has the capability to seamlessly resume existing test generationprocess upon interruption.

Keywords: P2R, Pairwise testing, resumption after interruption, test suite generation, checkpointing

1. Introduction

Software testing involves activities that are aimed toassess an attribute or capability of a program or system and determining whether or not it meets its required results and specification. To achieve acceptable level of quality, it is often desirable to test every possible combination of inputs under various configurations. Nevertheless, considering all exhaustive testing is practically impossible especially when dealing with large number of parameter inputs. Costing factors, resource constraints as well as strict time-to-market deadlines are amongst the main factors that inhibit such consideration. Earlier work suggests that pairwise strategies can be effective [6, 16] to systematically sample the parameter inputs (and generate a complete pairwise test suite) for many classes of practical system. For these reasons, myriad ofuseful pairwise strategies and their implementations have been developed both as research prototypes and commercial applications.

In line with customer demand for enhanced software functionalities (and complexities), pairwise strategies increasingly need to be able to sample potentially large number of parameter inputs resulting into significantly long running transaction[26, 29, 8]. Here, in the case of interruption (e.g. due to power failure or hardware malfunctions), there is a need to restart the whole generation process again from the beginning. If the pairwise test suite is significantly large and the overall progress is near completion, restarting means a lot of wasted efforts as far as resources and time are concerned. In order to address this issue, we have developed a new pairwise generation strategy, called P2R, capable of resuming its execution upon interruption. P2R serves as our research vehicle to investigate the adoption of checkpointing technique into a pairwise test generation strategy.

The rest of the paper is organized as follows. Section 2 outlines the related work on pairwise testing and checkpointing. Section 3 illustrates the design of P2R. Section 4 highlights our experimental and benchmarking results. Finally, section 5 gives our conclusion.

2. Related Work

As elaborated earlier, P2R strategy involves the amalgam between existing work on pairwise strategies and checkpointing techniques. As such, the related work shall cover both literatures.

Concerning related work on pairwise strategies, Lei et al classifies existing strategies in to algebraic and computational approach respectively [15]. The former approach uses predefined mathematical functions and lookup tables to generate pairwise test suites [15]. As such, strategies adopting algebraic approach often require fast generation time. In the absence of exact mathematical functions and lookup tables, such strategies are not applicable for large configurations[15, 31].

The later approach relies on the generation of the all pair combinations. Based on all pair combinations, the predefined search algorithm iterates the combinations space to generate the required test case until all pairs are covered. Although the searching process can be time consuming depending on the configuration, strategies adopting this approach can address the support for large configuration[1, 14].A significant number of computational based test suite generation algorithms have been developed, some are deterministic (same input parameter model leading to different test suites [13]) such as In-Parameter

Algorithm (IPO) [16] and Allpairs [2] algorithm while others are nondeterministic such as Automatic Efficient Test Generator(AETG) [5] and its variants. Some depend on greedy algorithms like Generic algorithm (GA) and Ant Colony Algorithm (ACA) [25] while others depend on heuristic search techniques such as Simulated Annealing (SA) [31].

AETG [5] and its variant AETGm [7], employ a greedy algorithm to construct the test case. In this strategy, each test covers as many uncovered combinations as possible. AETG uses the random search algorithm, thus the generated test case will be nondeterministic. Other AETG variants using stochastic greedy algorithms include: GA and ACA [25]. Although they share the common characteristic of being nondeterministic, The GA and ACA sometimes give better solutions than the original AETG. However GA, ACA and other artificial intelligence based strategies are limited when addressing large configurations.

IPO [16] strategy builds a pairwise test set for the first two parameters, and extends it to cover the first three parameters. Then the strategy continues to extend the test set until building a pairwise test set covering all the parameters. Apart from being deterministic in nature, covering one parameter at a time makes the IPO strategy less complex than AETG. Another example of strategies based on computational approaches is AllPairs [2].Similar IPO, AllPairs strategy shares the same property in producing deterministic test cases.

Regarding non-greedy strategies, some approaches adopt heuristic search techniques such as Simulated Annealing (SA) [31]. Simulated Annealing strategies start with some known test set. A series of transformations are then iteratively applied to cover all the pairwise combinations. Unlike AETG, IPO and AllPairs strategies, which build a test set from scratch, heuristic search techniques can predict the known test set in advance. However, there is no guarantee that the test sets produced are optimal.

Schroeder and Korel [24] develop a combinatorial strategy based on computational approach, relying on the input and output relationship. The strategy achieves reduction when some of the input parameters are known to have insignificant effect on the system (i.e. as don't care values).In this case, the strategy randomly assigns values to the don't care parameters in order to achieve reduction.

Jun and Jian proposed a Backtracking algorithm and search heuristics [31]. Despite its advantages, the strategy is restricted to small configurations. This is due to the long execution time caused by its dependence on exhaustive search methods. Inspired by Jun and Jian's proposal, G2Way[14] has been developed as a flexible heuristic that does not rely on exhaustive search methods. G2Way goes through the uncovered pairs and recombines them to form complete test suites. G2Way proved to be advantageous over many earlier strategies, in terms of generated test suite size and execution time.

The aforementioned strategies achieved their anticipated objectives of being efficient in execution time and test suite size. Although useful, earlier work has not sufficiently dealt with resumption capability (i.e. in order to support recovery upon interruption during test suite generation processes). Often, generating pairwise test data with large numbers of parameters can be time consuming, hence, susceptible to interruption. In this manner, there is a need for a pairwise strategy to address resumption.

Resumption consideration for a long running transaction is not new [18, 20]. Here, existing approach typically adopts checkpointing approach to tolerate undesired interruption by rolling back the system to a pre-interruption state. Briefly, checkpointing works by introducing strategic checkpoints during where execution status and generated data are stored on non-volatile memory [29]. If there is interruption during execution, system rolls back to the most recently passed checkpoint where execution is resumed. However, applying checkpointing is not without a cost. Typically, checkpointing requires holding the system execution state and data in some form of persistence storage to permit recovery which consumes time.

Checkpointing interval determines the flexibility of the recovery system [23]. According to the way checkpoints are located (or generated), checkpointing can be either placed periodically or dynamically [20, 21]. Periodic checkpointing saves system state either after the completion of event execution or after a fixed number of event completions [21] or can be triggered by a timer interrupt [20]. Here, choosing the optimal fixed frequency of checkpointing is the main challenge for periodic checkpointing [21, 3]. Related work on periodic checkpointing includes that of Chandy et al. [3], Gelenbe and Derochette [10], and Gelende [9]. They propose periodic checkpointing models for database recovery as well as investigate the optimal checkpoint interval that maximize the system availability and minimize the checkpointing overhead. Dynamic checkpointing have also been addressed in many studies [17] such as, Nicola and Van Spanje [19], Goes and Sumita [12] and Goes [11]. Using dynamic checkpointing, programmer need to divide the program into tasks in order to estimate the execution time, recovery time, and checkpointing time for each task [4]. In this case, checkpoint interval is dynamically determined based on some heuristics (e.g. execution time for each task of interests). There is also existing work that adopts both static and dynamic checkpointing such as that of Chandy and Ramamoorthy [4], Toueg and Babaoglu [26] and Upadhyaya and Saluja [27, 28].

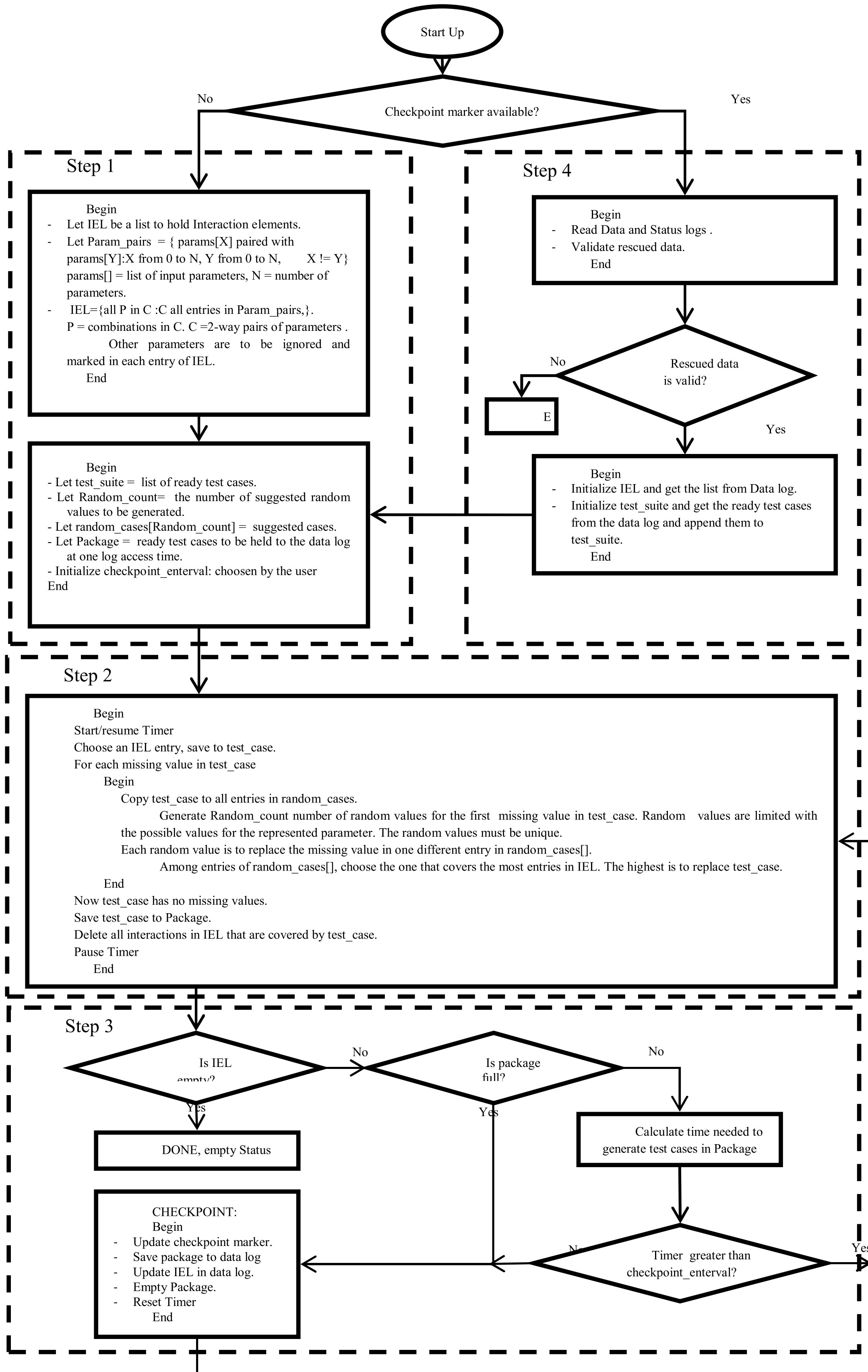

Fig 1. P2R Flowchart

3. The P2R Strategy

P2R generates test suites by suggesting a number of random values to complete each test case. The suggested values are then tested to know which case covers the largest number of uncovered pairs. The one that covers most is to be chosen as a test case, and then added to the test suite. While executing test suite generation, the process passes through a number of checkpoints where progress indicator and generated data are to be saved on a stack. When the system is resumed after interruption, P2R checks the previous session in order to verify completion status, if not, P2R prepares the needed data to be restored in order to resume the execution from the nearest possible point. Fig. 1 summarizes P2R strategy:

- **Step 1: Initialization and *Interaction Elements List (IEL)* generation.**

In this step, P2R generates the interaction elements list which represents pair interaction of the defined parameters. Specifically, the interaction elements list is a list of all possible 2-way combinations. Generating the 2-way interaction element list requires generating all possible combinations within each pair of input parameters at a given time.In this case, the algorithm decides which pair of parameters to be considered when finding all possible combinations between them.Each parameter is paired with all other parameters. Once paired, they are to be the significant pair, and all possible combinations between them are to be generated. The algorithm assigns a don't-care value (such as -1) to the rest of the parameters (at that time). For example, consider a system with 3 parameters (P0, P1 and P2).each parameter has two possible values (V0 and V1), the generated interaction element list is shown in Table 1.

Table 1: Example of IEL

P0	P1	P2
-1	V0	V0
-1	V0	V1
-1	V1	V0
-1	V1	V1
V0	-1	V0
V0	-1	V1
V1	-1	V0
V1	-1	V1
V0	V0	-1
V0	V1	-1
V1	V0	-1
V1	V1	-1

Step 2: The Test Suite Generation

In this step, P2R generates the test suite by suggesting a number of random values to fill the don't-care values (-1) in the interaction element list. The algorithm checks the number of covered pairs by each of the random values, and then, the random value with the highest coverage is chosen to be test case.

The algorithm starts with the first interaction element and keeps filling the missing values then checking the coverage. The chosen test case is added to the test suite and the covered interactions are deleted from the interaction element list. The process is continued until the interaction element list is empty. Hence, the complete coverage of all possible pairwise combinations is assured.

Step 3: Checkpointing

As highlighted in earlier section, checkpointing involves storing execution state based on some static or dynamic intervals in non-volatile memory. P2R support both static and dynamic intervals in three configurations. Firstly,P2R can locate static checkpoint intervals by holding each single generated test case at one log access. Secondly, P2R can also locate static checkpoint intervals by holding a list of generated test cases at one log access. Finally, P2R can also locate dynamic checkpoint intervals by holding the generated test cases when time needed to generate them lasted longer than a fixed time interval. The first configuration is enabled by setting the package size to be 1. Here, package will be full after each test case is generated, the latest version of theinteraction elements list will be saved. For the second configuration, choosing a different package size will decide

how many generated test cases are to be saved together in one log access. The third configuration is enabled by choosing the shorted possible checkpoint time interval and that time is saved.

Fig.2 and Fig.3 illustrate how the location of interruption can affect test generation in case when each ready test case is ckeckpointed after its completion or in case when each number of ready test cases are checkpointed together (either when they are limited by the count or by execution time).Fig. 2, shows a case where checkpoint is employed after generating each test case. The green colored cases are retrievable and the red colored cases are not yet generated.

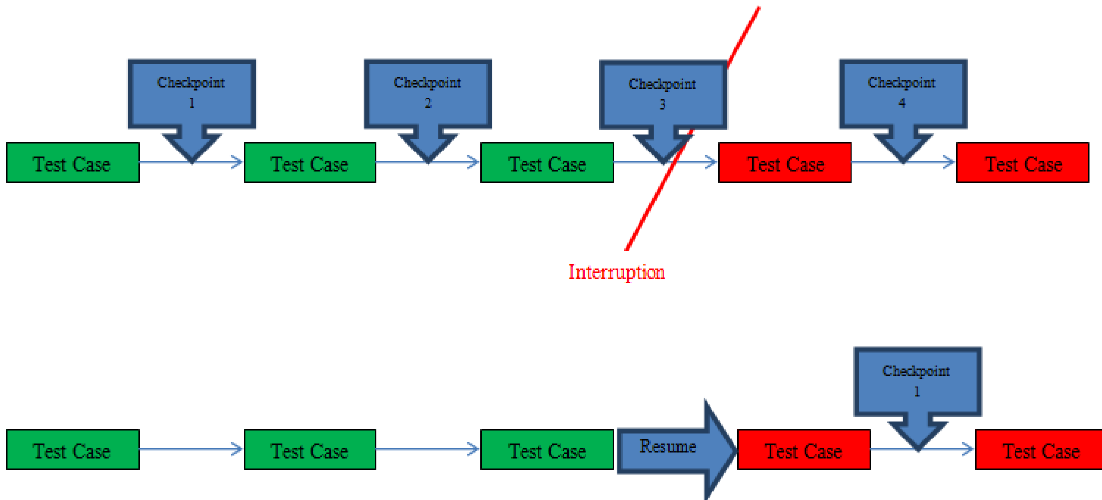

Fig 2.Interruption and resumption with one log access for each ready test case

Fig. 2 shows that each generated test case is retrievable no matter where the interruption occurs, however this system requires the maximum number of checkpoints, hence, it causes the maximum log access time. However, when system is resumed, time needed to reach interruption point is guaranteed to be as short as possible. Fig. 3, shows a resumption system that assigns a resumption point after each couple of generated test cases. The values of each two generated test case are rescued at once. The green colored cases are retrievable while the yellow colored cases are not. The red colored cases are not yet generated. Fig. 3 also applies on a system that places a checkpoint after execution time reaches minimal time interval.

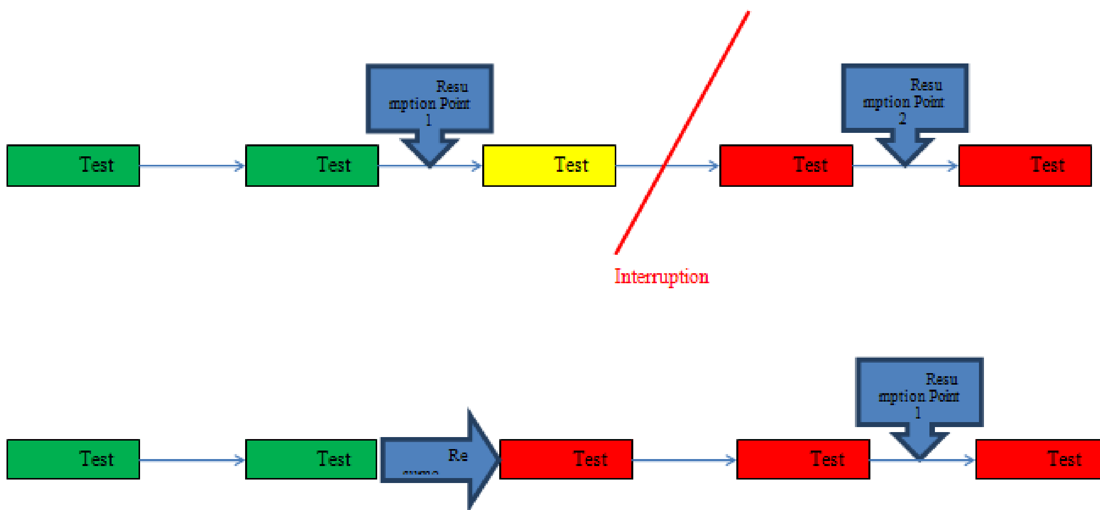

Fig 3.Interruption and resumption with one log access foreach two ready test case

From Fig. 3, test case 3 has not been rescued since the system resumes execution from the nearest resumption point; hence, test case 3 is to be regenerated when system is resumed. Advantageously, this system requires less log access time than the previous system.

To permit flexibility, P2R enables software testing engineers to manually choose the frequency of checkpoint by choosing the number of ready test cases to be shipped at one log access. The size of the package is the number of ready test cases to be log

accessed, in other words, the size of the package decides the frequency and count of checkpoints. Moreover, P2R enables users to set a minimal time interval for checkpoint frequency.

Step 4: System Restore

At this step, P2R is prepared to resume the previously interrupted test suite generation process. In this case, the P2R accesses the status log and the data log and checks the validity of the held data. Here, data log contains the previously generated test case and the current IEL. Based on these informations, P2R is able to identify the location of the interruption. Once system is resumed, rescued test cases are saved to the test suite list and the final available version of IEL is to be saved to IEL list. There, P2R is ready to continue the test suite generation process.

4. P2R Evaluation

In order to evaluate P2R, P2R's test suite generation process and P2R's checkpointing algorithm must be questioned separately.The test suite generation process must be examined whether it works correctly (generate correct data) and a benchmarking with other existing pairwise test generation algorithms based on execution time and generated test suite size is needed to decide the applicability of the algorithm. Checkpointing configuration for test suite generation considering time overhead and rollback time overhead are the basis of choosing between checkpointing interval. Other issue to be examined when evaluating P2R is the effectiveness of its checkpointing configuration. In order to achieve this evaluation, a number of experiments are performed and they will be highlighted in the next paragraphs.

4.1 P2R checkpointing comparison (based on checkpointing time overhead and rollback time overhead)

Time overhead caused by checkpointing must be considered when applying checkpointing. Owing to the fact that this time overhead is mainly caused by the log access during checkpointing, reducing the frequency of checkpointing can reduce the log access time overhead when executing P2R. However, reducing frequency of checkpointing will increase rollback time overhead, as will be proven later. While choosing the optimal frequency of checkpointing, checkpoint time overhead and rollback time overhead must be considered as two contradicting factors. As highlighted earlier, P2R enables checkpointing in three configurations:

- Configuration 1: checkpoint each generated test case.
- Configuration 2: checkpoint each X number of generated test cases (X is an integer number).
- Configuration 3: checkpoint when time needed to generate a test case or a number of test cases is more than Y seconds (Y represents time interval), a checkpoint is placed in case more than X number of test cases are generated before time interval Y expires (X is an integer number).

Another issue to be considered when choosing the frequency of checkpoints is the fact that time needed to generate test cases is not uniform; some test cases require much longer generation time than others. P2R generates test cases by suggesting a number of random values to replace missing values in the interaction elements list (IEL), then it choose the random number with highest coverage of interactions in the interaction elements list. This process requires multi access to mentioned list, thus, the larger IEL is, the longer time required to generate a test case. After choosing the best values for the test case, covered interactions in IEL are deleted. While system is generating test cases, IEL becomes shorter, and so, time needed to generate the next test cases becomes shorter and shorter. To illustrate how time needed to generate test cases differs from while executing a test suite generation, a simple experiment have been performed: test suite generation for three systems with large configuration (System A, System B and System C) is executed and time needed to generatetest cases for each system is measured and shown in Fig. 4. The tested systems are systems with larger configurations commonly used to evaluate test suite generation algorithms. The test systems are:

- System A: Ten parameters each with ten possible input values.
- System B: Ten parameters each with fifteen possible input values.
- System C: Twenty parameters each with ten possible input values.

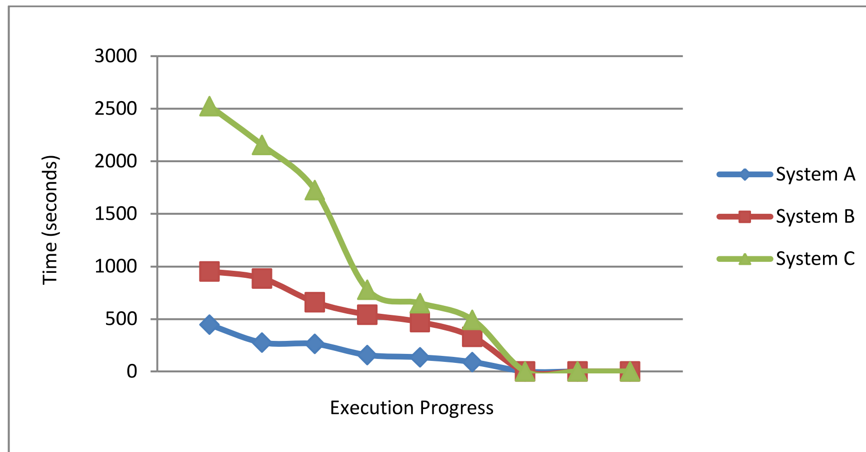

Fig 4. Test case execution time against execution progress

From Fig. 4, time needed to generate each test case decreases while the progress is going forward. This fact that generation time is dynamic should not be ignored while choosing the locations of the checkpoint. If interruption occurred after generating the 9th test case after the last checkpoint, and time needed to generate each of the last 9 test cases was long, then interruption will cause regeneration for the mentioned test cases. Another case, assume a system that checkpoints every ready test case, then when the system execution is nearly to completion, log access will be very frequent and checkpoint time overhead will be high. Providing the system with the ability to dynamically decide whether a checkpoint is to be placed when time needed to generate a test case is so high can avoid much checkpoint latency, and still time needed to resume execution is likely to be less than in the case of checkpointing every fixed number of generated test cases.

In order to highlight the difference in performance between the three mentioned configurations, the count of checkpoints and checkpointing time overhead for each configuration is to be compared. Moreover, rollback time overhead from different interruption locations is to be compared. Firstly, we start testing checkpoint time over head for P2R executions with three configurations:

- Configuration 1: checkpoint each ready test case.
- Configuration 2: checkpoint every ten ready test cases
- Configuration 3: checkpoint every ten ready test cases except if test case generation required more than 1 minute, then place a checkpoint.

P2R is to be executed using the mentioned configurations to generate test suites for five test suite generation processes, time overhead for each configuration running each tested systems are illustrated in Fig. 5.and the count of generated checkpoints are illustrated in Fig. 6. The examined systems are commonly used to evaluate test suite generation algorithms, the tested system are:

- S1: Ten parameters each with five possible input values.
- S2: Ten parameters each with ten possible input values.
- S3: Thirteen parameters each with three possible input values.
- S4: Ten parameters each with fifteen possible input values.
- S5: Twenty parameters each with ten possible input values.

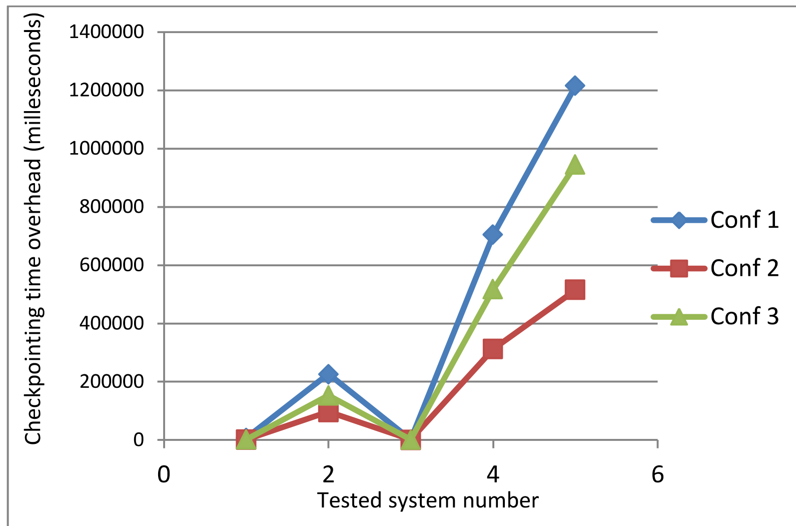

Fig 5. Checkpoint time overhead (milliseconds) for different P2R configurations running test suite generation for different systems

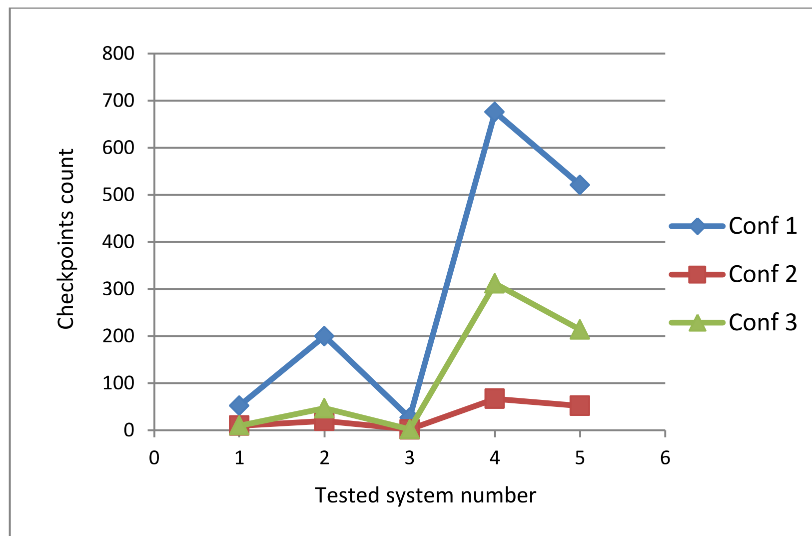

Fig 6. Count of generated checkpoints for different P2R configurations running test suite generation for different systems

Based on Fig. 5, checkpointing time overhead is the highest when checkpointing is done after each every ready test case, while the mentioned time overhead is the lowest when checkpointing is done every ten ready test cases. This is because checkpointing in the first configuration generated the most number of checkpoints as Fig.6 shows. It can be noticed that the third configuration's checkpoint time overhead is intermediate to the first and the second configurations, this is because configuration 3 behaves almost like the first configuration in the beginning of the execution (when test case generation lasts long time) and then it begins to behave more like the second configuration when test case generation begins to take shorter time.

When testing rollback time overhead, interruption must be forced at some chosen stages of execution. Choosing the location of interruption can give different results, in order to find the best locations that highlight the difference in rollback time overhead of checkpointing configurations, interruption is to be forced at different locations after system is resumed. Time latency when resumed is to be measured and compared between the three configurations.

Suggested locations of interruption points:

- Interruption point 1: After generating the first test case, generating the first test case is likely to require the longest execution time, thus, comparing system behavior when interruption occurs after generating this test case using the three configurations can show big difference in rollback time overhead between them.
- Interruption point 2: After generating the ninth test case, this interruption is meant to show the maximum difference in rollback time latency between the three configurations; the first nine test cases require the longest time execution for P2R. Especially for the second configuration, rollback time overhead is expected to be the highest.
- Interruption point 3: After generating the tenth test case, this interruption point might not show difference in rollback time overhead.

- Interruption point 4: One test case before the last generated test case, rollback time overhead is expected to be the lowest at this stage, the difference might not be significant.
- Interruption points 5,6 ,7,8,9 and 10: Random interruption points when execution reaches 10%,20%,30%, 50% ,70%, and 90% of completion, interruptions normally occur randomly, performing interruption after each test case generation might not be practical. Choosing such random point can still show the difference between the configurations.

P2R with the three mentioned configurations is executed to generate test suites for a system with ten input parameters each with ten possible values, while execution, 10 interruption points will be forced to the system (the same mentioned interruption locations). After each interruption, system is resumed and time needed to reach the last generated checkpoint is measured. Fig. 7 shows the difference in rollback time overhead, since the difference in time latency is so high, the figure doesn't show the tie differences, actually most of the values appear to be zero on the graph, thus the results are shown again in Table 2.

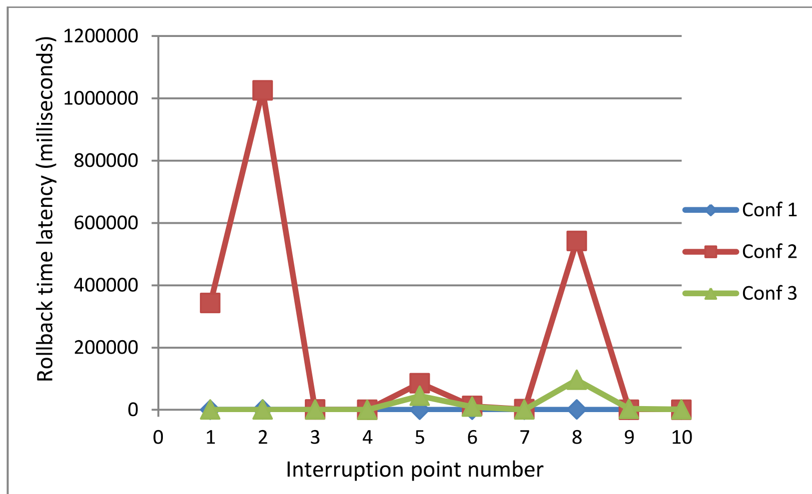

Fig 7. Rollback time overhead for P2R in different configurations.

According to Fig. 7 and Table 2, the second configuration is very likely to cost the highest rollback time overhead and the first configuration is almost always with the least rollback time overhead. The third configuration has shown good results in both performed tests, since the maximum possible rollback latency cannot be higher than two minutes and backup is not done as frequent as the first configurations. The fact that test cases differ in generation time made the third configuration appear to be better than the other two configurations, applying this configuration is recommended which is what the mentioned examinations have proven.

Table 2. Rollback time overhead for P2R in different configurations

Interruption location	Configuration 1	Configuraion 2	Configuration 3
1	510	343215	825
2	571	1025429	554
3	614	584	745
4	412	450	385
5	450	85412	45125
6	377	12053	10225
7	365	1541	589
8	625	542156	96215
9	354	456	3251
10	421	354	1453

4.2 Correctness Inspection

In order to investigate the correctness of P2R, two examinations must be done, one to confirm the correctness of the test suite generation and the other to confirm the correctness of the checkpointing technique. Investigating the correctness of the test suite generation is done by checking the 2-way coverage of the generated test suite. P2R have been executed to generate many test suite generations, and the 2-way coverage has been confirmed.

Testing the correctness of P2R checkpointing was performed by forcing interruptions of test suite generation process in every stage of the generation process. After interruption is done, system is to be restarted in order to study the behaviour of the system. In order to assure that interruption is tried in every stage of the execution, a halting code line (ex. a read line command) must be added in each stage. When the halt state is reached, the system is to be restarted.

When checkpointing is done, the needed data to enable resumption is stored on a special stack. The stacks are to be examined whether it held the correct data or not. In order to test the resumption process, the system is to be interrupted after generating a number of test cases, for example 20. The location of interruption and the generated data are to be checked whether stored correctly in the logs. When restarting the system, the test suite generation is continued till the end, the first 20 entries in the outcome must be identical to the previously backed up 20 entries. Since P2R is a non-deterministic strategy, any change in the first 20 sets means the resumption have failed.

4.3Effectiveness Inspection

In order to question the effectiveness of P2R, a long time consuming systems are interrupted after the generation of a number of test cases. Time needed to generate the ready cases is to be measured, and when system is resumed, time needed to reach the same execution progress is also measured and compared with the execution time. The testedsystems are:

- S1: Ten 10-valued parameters.
- S2: Twenty 20-valued parameters.

System S1 has been interrupted after the generation of 20 test cases, time needed to generate the 20 sets was 17 minutes. When restarted, 520 ms were needed to reach the interrupted point. And System S2 has been interrupted after 60 hours of execution. 60 hours were needed to generate 27 test cases. The time needed to pursue the generation process was 5392 ms.

According to the mentioned experiment, the time saved by the ability of resumption was very long. Restarting the generation process is so much time and effort consuming.

4.4. Comparison with other pairwise testing strategies

In order to investigate the efficiency of the P2R in terms of test suite sizes and time consumption, a comparison between P2R and other existing pairwise testing strategies has been done. P2R has been compared with AETG, AETGm, In Parameter Order(IPO), Simulated Annealing (SA), Generic Algorithm (GA), Ant Colony Algorithm (ACA), All Pairs and G2Way. Each system is tested by generating a test suite for each of the suggested system:

- S1: 3 3-valued parameters
- S2: 4 3-valued parameters,
- S3: 13 3-valued parameters,
- S4: 10 10-valued parameters,
- S5: 10 15-valued parameters,
- S6: 10 5-valued parameters
- S7: 1 5-valued parameters, 8 3-valued parameters and 2 2-valued parameters.

Table 4 and Table 5 shows comparison between the mentioned algorithms in term of generated test suite sizes and the required time to finish the execution, all entries in the mentioned tablesare taken from [32] except for Allpairs, G2Way and P2R. Tables entries related to Allpairs and G2Way have been taken from [14]. Tables entries related to P2R have been taken from experimental results performed on our platform.

Table 4. Comparison in term of generated test suite size.

System	AETG	AETGm	IPO	SA	GA	ACA	All Pairs	G2Way	P2R
S1	NA	NA	NA	NA	NA	NA	10	10	9
S2	9	11	9	9	9	9	10	10	9
S3	15	17	17	16	17	17	22	19	27
S4	NA	NA	169	NA	157	159	177	160	200
S5	NA	NA	361	NA	NA	NA	390	343	675
S6	NA	NA	47	NA	NA	NA	49	46	50
S7	19	20	NA	15	15	16	21	23	24

Table 5. Comparison in term of execution time (in seconds)

System	AETG	AETGm	IPO	SA	GA	ACA	All Pair	G2Way	P2R
S1	NA	NA	NA	NA	NA	NA	0.08	0.047	0.013
S2	NA	NA	NA	NA	NA	NA	0.23	0.062	0.016
S3	NA	NA	NA	NA	NA	NA	0.45	0.25	155
S4	NA	NA	03	NA	866	1180	5.03	2.906	4637
S5	NA	NA	0.72	NA	NA	NA	10.36	1.753	52165
S6	NA	NA	0.05	NA	NA	NA	1.02	0.687	582
S7	NA	58	NA	214	22	31	0.35	0.33	67

AETG, AETGm and SA have been applied on an Intel Pintium IV 1.8 Ghz, with Linux operating system. The tool have been developed using C++ programming language [32]. While IPO have been applied on an Intel Pintium II 450 Mhz with Windows XP operating system. IPO was programmed using Java programming language [32]. CA and ACA where applied on an Intel Pintium IV 2.26 Ghz with Windows XP operating system and C programming language has been used to develop it [32]. Allpairs and G2Way have been applied on an Intel Pintium IV 1.8 Ghz with Windows Vista operating system [14]. Allpairs have been developed using Perl programming language while G2Way have been developed using C++ programming language [14]. P2R have been developed using Java programming language and its been applied on an Intel Core I3 1.33 Ghz with Windows 7 operating system.

Results related to P2R in Table 4 and Table 5, are generated when P2R is running with the following configurations:

- P2R suggests three random values for each parameter when generating a test case.
- P2R generates a checkpoint every ten ready test cases, unless the last checkpoint have been passed later than two minutes, then it places a checkpoint.

Based on Table 4 and Table 5, other pairwise test suite generation techniques yielded better results than P2R. One reason is that P2R puts significant emphasis on recovery performance and not on test size performance. On a positive note, P2R can improve its generated test size by enabling more iteration between randomized values but with the expense of execution time.

5. Conclusion

A new test suite generation strategy, P2R, has been proposed. The strategy has efficiently and effectively saved much effort and time in case of process interruption. As part of future work, we are considering the support for t-way test generation and resumption as part of P2R enhancement.

References

1. A. A. Alsewari dan K. Z. Zamli, "A Harmony Search Based Pairwise Sampling Strategy for Combinatorial Testing," International Journal of the Physical Sciences, vol. 7, no. 7, pp. 1062 - 1072, 212.
2. J. Bach, "Allpairs Test Case Generation Tool". Available: http://tejasconsulting.com/opentestware/feature/allpairs.html.
3. D. Cohen, S. Dalal, M. Fredman dan G. Patton, "The AETG System: An Approach to Testing based on Combinatorial Design," IEEE Trans. on Software Engineering, vol. 23, no. 7, pp. 437 - 444, 1997.
4. D. Cohen, S. Dalal dan G. Patton, "The Combinatorial Design Approach to Automatic Test Generation," IEEE Software, vol. 13, no. 5, pp. 83 - 88, 1996.
5. M. B. Cohen, Designing Test Suites for Software Interaction Testing, PhD Thesis, University of Auckland, 2004.
6. P. Danziger, E. Mendelsohn, L. Moura dan B. Stevens, "Covering Arrays Avoiding Forbidden Edges," Theoretical Computer Science, vol. 410, no. 52, pp. 5403 - 5415, 2009.
7. E. Gelenbe dan D. Derochette, "Performance of Rollback Recovery Systems Under Intermittent Failures," ACM, vol. 21, no. 6, pp. 493 - 499, 1978.
8. E. Gelenbe, "On the Optimum Checkpoint Interval," Journal of the ACM, vol. 26, no. 2, pp. 259 - 270, 1979.

9. P. B. Goes, "A Stochastic Model for Performance Evaluation of Main Memory Resident Database Systems," ORSA Journal on Computing, vol. 7, pp. 269-282, 1995.
10. P. B. Goes dan U. Sumita, "Stochastic Models for Performance Analysis of Database Recovery Control," IEEE Trans. on Computer, vol. 44, pp. 561 - 576, 1995.
11. M. Grindal, J. Offutt dan S. F. Andler, "Combination Testing Strategies: A Survey," Software Testing, Verification, and Reliability, vol. 15, pp. 167-199, 2004.
12. M. F. Klaib, K. Z. Zamli, N. A. M. Isa, M. I. Younis dan R. Abdullah, "G2Way A Backtracking Strategy for Pairwise Test Data Generation," in 15th Asia-Pacific Software Engineering Conference, 2008, Beijing, 2008.
13. Y. Lei, R. Kacker, D. R. Kuhn, V. Okum dan J. Lawrence, "IPOG: A General Strategy for T-Way Software Testing," in 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, Tucson, AZ, 2007.
14. Y. Lin, B. R. Preiss, W. M. Loucks dan E. D. Lazowska, "Selecting the Checkpoint Interval in Time Warp Simulation," in 7th Workshop on Parallel and Distributed Simulation, New York, NY, 1993.
15. V.F. Nicola, "Checkpointing and the Modeling of Program Execution Time," *Software Fault-Tolerance,* M.R. Lyu, ed., pp. 167-188. John Wiley, 1995.
16. J. S. Plank, M. Beck, G. Kingsley dan K. Li, "Libckpt: Transparent Checkpointing under Unix," in Usenix Winter 1995 Technical Conference, New Orleans, LA, 1995.
17. R. Radhakrishnan, T. McBrayer, K. Subramani, M. Chetlur, V. Balakrishnan dan P. Wilsey, "A Comparative Analysis of Various Time Warp Algorithms Implemented in the WARPED Simulation Kernel," in the 29th Annual Simulation Symposium, 1996.
18. A. Ranganathan dan S. Upadhyaya, "Simulation Analysis of a Dynamic Checkpointing Strategy for Real-Time Systems," in 27th Annual Simulation Symposium, La Jolla, CA, 1994.
19. S. A. Reddy, "Dynamic Interval Determination for Page Level Incremental Checkpointing," National Institute of Technology, Rourkela, India, 2007.
20. P. Schroeder dan B. Korel, "Black-box Test Reduction Using Input-Output Analysis," in ACM SIGSOFT International Symposium on Software Testing and Analysis, New York, NY, 2000.
21. T. Shiba, T. Tsuchiya dan T. Kikuno, "Using Artificial Life Techniques to Generate Test Cases for Combinatorial Testing," in 28th Annual International Computer Software and Applications Conference, Hong Kong, 2004.
22. S. Toueg dan Ö. Babao?lu, "On the Optimum Checkpoint Selection Problem," SIAM Journal on Computing, vol. 13, no. 3, pp. 630 - 649, 1984.
23. J. S. Upadhyaya dan K. K. Saluja, "A Watchdog Processor based General Rollback Technique with Multiple Retries," IEEE Trans. on Software Engineering , vol. 12, no. 1, pp. 87 - 95, 1986.
24. S. Upadhyaya dan K. Saluja, "An Experimental Study to Determine Task Size for Rollback Recovery Systems," IEEE Trans. on Computers, vol. 37, no. 7, pp. 872 - 877, 1988.
25. N. Vaidya, "Impact of Checkpoint Latency on Overhead Ratio of a Checkpointing Scheme," IEEE Trans. on Computers, vol. 46, no. 8, pp. 942 - 947, 1997.
26. Y. Xun, C. M.B. dan M. A.M., "GUI Interaction Testing: Incorporating Event Context," IEEE Trans. on Software Engineering, vol. 37, no. 4, pp. 559 - 574 , 2011.
27. J. Yan dan J. Zhang, "Backtracking Algorithms and Search Heuristics to Generate Test Suites for Combinatorial Testing," in 30th Annual International Computer Software and Applications Conference, Chicago, IL, 2006.
28. M. Younis, K. Z. Zamli dan N. Isa, "IRPS -An Efficient Test Data Generation Strategy for Pairwise Testing," in 12th International Conference on Knowledge-Based and Intelligent Information & Engineering Systems, Zagreb, Croatia, 2008.
29. Y. Lei dan K. Tai, "In-Parameter-Order: A Test Generation Strategy for Pairwise Testing," in 3rd IEEE International High-Assurance Systems Engineering Symposium, Washington, DC, 1998.
30. V. F. Nicola dan J. M. v. Spanje, "Comparative Analysis of Different Models of Checkpointing and Recovery," IEEE Trans. on Software Engineering, vol. 16, no. 8, pp. 807-821, 1990.
31. K. M. Chandy dan C. Ramamoorthy, "Rollback and Recovery Strategies for Computer Programs," IEEE Trans. on Computer, vol c.21, no. 6, pp. 546 - 556, 1972.
32. K. Chandy, "A Survey of Analytic Models of Rollback and Recovery Strategies," IEEE Computer, vol. 8, no. 5, pp. 40 - 47, 1975.