Mutation Operators used in MERLIN

Nigar Azhar Butt, Muhammad Uzair Khan, Muhammad Zohaib Iqbal

December 20, 2023

1 Mutation Operators for Endless Runner Games

In this section, we propose mutation operators based on common faults encountered in endless runner games. Mutation operators represent common mistakes and errors made by programmers, are intentionally introduced into a program[1]. These operators make small syntactic modifications to the original program, generating a set of mutant versions. The purpose of employing mutation analysis stems from the challenge of locating real programs with known faults. Consequently, this methodology is widely utilized by software testing researchers to assess the applicability and efficacy of their testing techniques[1].

We started our efforts to design mutation operators by investigating commonly occurring faults. Mutation analysis relies on the utilization of mutation operators, which are sets of syntactic rules employed for modifying a program or artifact. The effectiveness of tests heavily depends on the quality of these mutation operators. Generally, mutation operators are established using fault models to define mutation operators, wherein each type of fault is leveraged to create specific instances of those faults through mutation operators[2]. Like, the muJava class-level operators were developed based on a previous fault model proposed by[3]. We have identified **12** mutation operators corresponding to **8** commonly occurring faults in context of games based on our previous work[4] (as shown in Table 1).

1.1 Reward Update Statement Deletion (RUSD)

The Reward Update Statement Deletion mutation operator deletes a statement that contains a definition or assignment to a reward variable such as score, health point, etc. It is similar to the traditional mutation operator *Statement Deletion* mutation operator described by Offutt in the context of Ada programs[5], however, this operator focuses specifically on the reward-related statements in game code. By deleting these statements, the operator introduces variations in how the rewards are calculated, acquired, or modified during gameplay (Figure 1a and 1b).

Table 1: Mutation operators for endless runner games based on commonly occurring faults

Commonly occurring faults	**Mutation Operator**	**Name**	**Traditional Operator**	**Description**
Incorrect Reward or Punishment Fault	Reward Update Statement Deletion	RUSD	Statement Deletion	Deletes a statement that contains definition or assignment to a reward variable such as score, health point etc.
	Reward Update Amount Replacememt	RUAR	Constant Replacement	Modifies the code by increasing the value of the reward update operation to a reward variable such as score, health point etc.
	Reward Update Operator Replacement	RUOR	Arithmetic Operator Replacement	Replaces the operator used to update or modify the reward variable value. It involve changing arithmetic operators such as +, -,* , /, %, etc.
Collision Detection	Disable collision detection	DCD	Statement Deletion	Deletes a statement that contains call to collision detection operation or is responsible for collision detection mechanism.

Table 1 continued from previous page

Commonly occurring faults	**Mutation Operator**	**Name**	**Traditional Operator**	Description
Unresponsive Action Fault	Disable Action Listener	DAL	Statement Deletion	Deletes the statement that contains action listener for an in-game action such as jump, move left right etc.
Incorrect Response Fault	Action Response Replacement	ARR	Statement Replacement	Replaces an action response with a different action response.
Repeated Action Inconsistency	Action Disablement with Delay	ADD	Statement Insertion	Modifies action listener response code to introduce a delay or condition that prevents the action from being executed again immediately after initial execution.
Accelerated Response	Increase Action Velocity	AVI	Constant Value Replacement	Increases the action velocity by multiplying n where n >1 to the definition and assignment of action velocity
Delayed Response	Decrease Action Velocity	AVD	Constant Value Replacement	Decreases the action velocity by multiplying n where n <1 and n >0 to the definition and assignment of action velocity
Game Freeze	Game Freeze on action	GFA	Statement Insertion	Adds an infinite loop to the action listener, causing the game to freeze.
	Game Freeze at time	GFT	Statement Insertion	Adds an infinite loop to the game execution loop or update function, causing the game to freeze at time t.
	Game Freeze on Reward	GFR	Statement Insertion	Adds an infinite loop to the game execution loop or update function, causing the game to freeze when a particular reward value is reached.

1.2 Reward Update Amount Replacement (RUAR)

This mutation operator focuses on replacing the amount by which reward variable such as score, health point etc, are updated within the game code. This mutation operator is similar to the *Constant Value Replacement* mutation operator introduced by Offutt but applied specifically to reward updates in the context of games. The mutation operator works by modifying the code that updates the reward amount. It alters the definition or assignment of reward update by replacing the constant or variable responsible for the updated amount with one having a lesser or greater value (Figure 1c and 1d).

1.3 Reward Update Operator Replacement (RUOR)

This mutation operator focuses on replacing the operator used to update or modify the reward variable such as score, health point etc, in a game system. It involves changing arithmetic operators. For example, replacing '+' with '−', or '∗' with '/'. It is similar to the *Arithematic Operator Replacement* mutation operator described in the context of java programs[3], however, this operator focuses specifically on the reward-related statements in game code (Figure 1e and 1f).

1.4 Disable Collision Detection (DCD)

This mutation operator deletes a statement that contains call to collision detection mechanism or function. It is similar to the traditional *Statement Deletion* mutation operator described by Offutt in the context of Ada programs[5], however this operator focuses specifically on the collision detection-related statements in game code. Consider a game where a player controls a character that needs to avoid obstacles. The collision detection mechanism ensures that the character collides with obstacles, triggering appropriate game events or consequences.By deleting these statements, we disable the collision detection mechanism. As a result, the game will no longer react to collisions, potentially allowing the player to move through obstacles without any consequences (Figure 1g and 1h).

1.5 Disable Action Listener (DAL)

This mutation operator deletes a statement that contains action listener for in-game player actions such as `jump`, `fire` etc. It is similar to the traditional *Statement Deletion* mutation operator described by Offutt in the context of Ada programs[5], however this operator focuses specifically on the action listener-related statements in game code. When an action listener is disabled, the associated action or event will be ignored or not processed by the game. This means that any user input or trigger that would have invoked the action will have no effect. The game will not respond to that particular action until the listener is re-enabled (Figure 1i and 1j).

1.6 Action Response Replacement (ARR)

This mutation operator replaces the statement or statements that contain a response or effect of triggering an action listener for in-game player actions such as `jump`, `fire` etc; with statments from an alternative action listener. It is similar to the traditional *Statement Replacement* mutation operator[5], however this operator focuses specifically on the action listener-related statements in game code (Figure 1k and 1l).

1.7 Action Disablement with Delay (ADD)

This mutation operator aims to simulate the disabling of an action for a specific time after its initial execution. It involves modifying the code to introduce a delay or condition that prevents the action from being executed again immediately (Figure 1m and 1n). However, if the time for which the action has been disabled is too short, then the mutant appears to be equivalent to the end user. On average, individuals typically exhibit a reaction time of approximately 250 milliseconds when responding to visual stimuli. Through training, most people can improve their reaction time to a maximum of around 190-200 milliseconds [6]. Nvidia, the pioneering company behind graphics cards, claims that highly skilled gamers possess an even faster average reaction time of 150 milliseconds[6]. Hence, if the time is less than the reaction time then it may not even be noticed by player. It takes inspiration from statement insertion and condition modification mutation operators. It focuses on seeding *Repeated Action Inconsistency* fault[4] via a modification to the action listener which disables it for a time once an action is initially executed.

1.8 Increase Action Velocity (AVI)

This mutation operator focuses on increasing the action velocity. This mutation operator is similar to the *Constant Value Replacement* mutation operator introduced by Offutt but applied specifically to the variable responsible for action velocity in the context of games. The mutation operator works by modifying the code that is the assignment of action velocity by multiplying the constant or variable value with a number greater than 1 ($n > 1$). It alters the definition or assignment of action velocity by increasing the constant or variable responsible (Figure 1o and 1p). As stated in 1.7, if the time or speed variation falls under the reaction time then it may not seem like an issue or fault to the player.

1.9 Decrease Action Velocity (AVD)

This mutation operator focuses on decreasing the action velocity. This mutation operator is similar to the *Constant Value Replacement* mutation operator introduced by Offutt but applied specifically to the variable responsible for action velocity in the context of games. The mutation operator works by modifying the code that is the assignment of action velocity by multiplying the constant or variable value with a number less than 1 and greater than 0 ($0 < n < 1$). It alters the definition or assignment of action velocity by decreasing the constant or variable responsible (Figure 1q and 1r). Similarly to 1.8, if the time or speed variation falls under the reaction time then it may not seem like an issue or fault to the player.

1.10 Game Freeze on action (GFA)

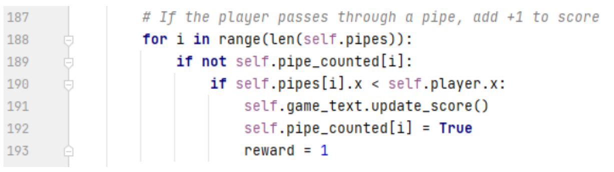
This mutation operator inserts a code-block to action listener that causes a game to freeze on action execution like adding an infinite loop to the action listener for in-game player actions such as *jump*, *fire* etc. It is similar to the traditional *Statement insertion* mutation operator described by Offutt in the context of Ada programs[5], however, this operator focuses specifically on the action listener-related statements in game code to add the required code-block. Hence, while the *Statement Insertion* operator adds a statement to the code, the **GFA** operator specifically adds an infinite loop to the action listener, causing the game to freeze (Figure 1s and 1t).

1.11 Game Freeze on time (GFT)

This mutation operator inserts a code-block to the game execution loop that causes a game to freeze at time t specified by the operator. It is similar to the traditional *Statement insertion* mutation operator described by Offutt in the context of Ada programs[5], however, this operator focuses specifically on the game execution loop (Figure 1u and 1v). For example, in case of Adding an infinite loop to randomly freeze game functionality in Unity3D, you could add the infinite loop in the Update function or a custom co-routine. The purpose of the **GFT** mutation operator is to evaluate how the game handles freezing or unresponsive behavior at a particular moment in time. By intentionally modifying the code to freeze the game at a specific time, we can assess if the game continues to function correctly and gracefully handles freezing scenarios.

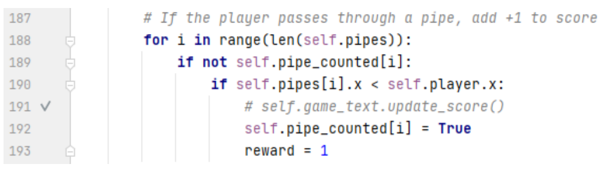
1.12 Game Freeze on Reward (GFR)

This mutation operator inserts a code-block to the game execution loop that causes a game to freeze on once reward specified by the operator has been achieved. It is similar to the traditional *Statement insertion* mutation operator described by Offutt in the context of Ada programs[5], however, this operator focuses specifically on the game execution loop and checks for reward achieved (Figure 1w and 1x).



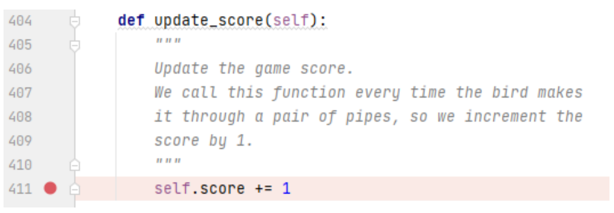
```python
# If the player passes through a pipe, add +1 to score
for i in range(len(self.pipes)):
    if not self.pipe_counted[i]:
        if self.pipes[i].x < self.player.x:
            self.game_text.update_score()
            self.pipe_counted[i] = True
            reward = 1
```

(a) original code



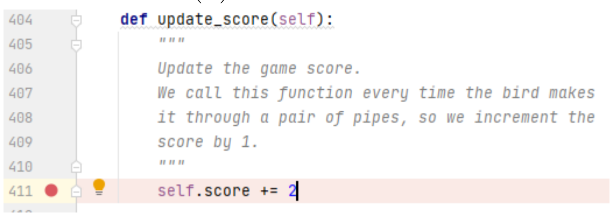
```python
# If the player passes through a pipe, add +1 to score
for i in range(len(self.pipes)):
    if not self.pipe_counted[i]:
        if self.pipes[i].x < self.player.x:
            # self.game_text.update_score()
            self.pipe_counted[i] = True
            reward = 1
```

(b) RUSD mutant



```python
def update_score(self):
    """
    Update the game score.
    We call this function every time the bird makes
    it through a pair of pipes, so we increment the
    score by 1.
    """
    self.score += 1
```

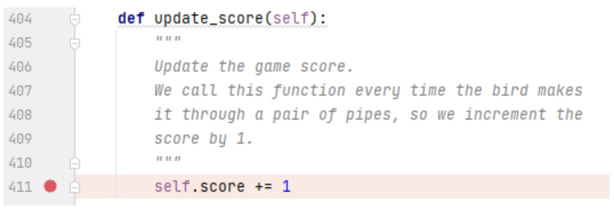
(c) original code

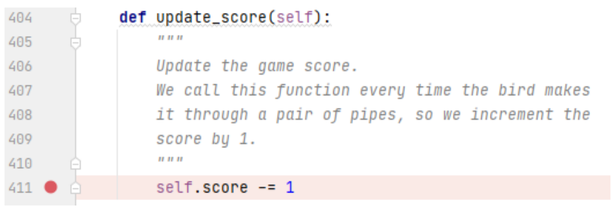


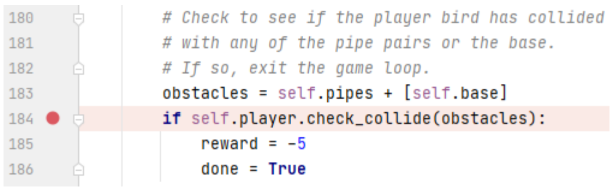
```python
def update_score(self):
    """
    Update the game score.
    We call this function every time the bird makes
    it through a pair of pipes, so we increment the
    score by 1.
    """
    self.score += 2
```

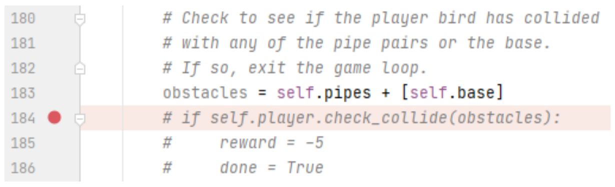
(d) RUAR mutant

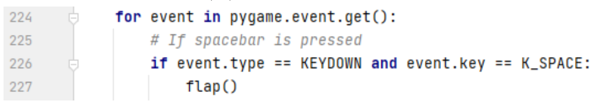
Figure 1: Examples of mutation operators (cont.).

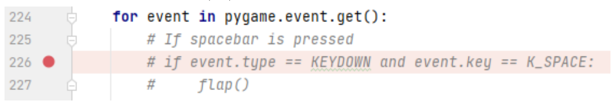

```python
def update_score(self):
    """
    Update the game score.
    We call this function every time the bird makes
    it through a pair of pipes, so we increment the
    score by 1.
    """
    self.score += 1
```
(e) original code

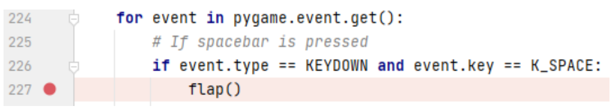

```python
def update_score(self):
    """
    Update the game score.
    We call this function every time the bird makes
    it through a pair of pipes, so we increment the
    score by 1.
    """
    self.score -= 1
```
(f) RUOR mutant

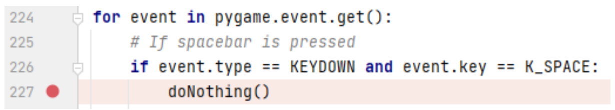

```python
    # Check to see if the player bird has collided
    # with any of the pipe pairs or the base.
    # If so, exit the game loop.
    obstacles = self.pipes + [self.base]
    if self.player.check_collide(obstacles):
        reward = -5
        done = True
```
(g) original code

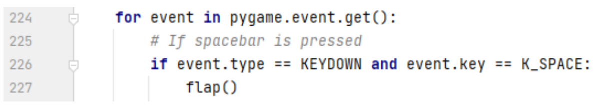

```python
    # Check to see if the player bird has collided
    # with any of the pipe pairs or the base.
    # If so, exit the game loop.
    obstacles = self.pipes + [self.base]
    # if self.player.check_collide(obstacles):
    #     reward = -5
    #     done = True
```
(h) DCD mutant

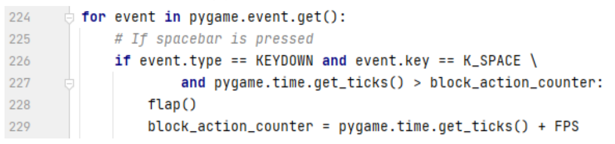

```python
for event in pygame.event.get():
    # If spacebar is pressed
    if event.type == KEYDOWN and event.key == K_SPACE:
        flap()
```
(i) original code

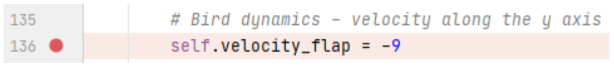

```python
for event in pygame.event.get():
    # If spacebar is pressed
    # if event.type == KEYDOWN and event.key == K_SPACE:
    #     flap()
```
(j) DAL mutant

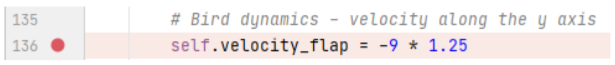

```python
for event in pygame.event.get():
    # If spacebar is pressed
    if event.type == KEYDOWN and event.key == K_SPACE:
        flap()
```
(k) original code

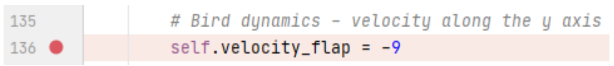

```python
for event in pygame.event.get():
    # If spacebar is pressed
    if event.type == KEYDOWN and event.key == K_SPACE:
        doNothing()
```
(l) ARR mutant

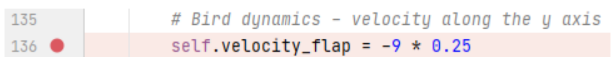

```python
for event in pygame.event.get():
    # If spacebar is pressed
    if event.type == KEYDOWN and event.key == K_SPACE:
        flap()
```
(m) original code

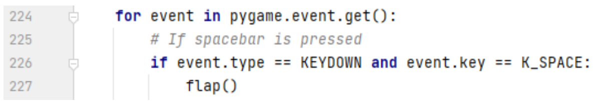

```python
for event in pygame.event.get():
    # If spacebar is pressed
    if event.type == KEYDOWN and event.key == K_SPACE \
            and pygame.time.get_ticks() > block_action_counter:
        flap()
        block_action_counter = pygame.time.get_ticks() + FPS
```
(n) ADD mutant

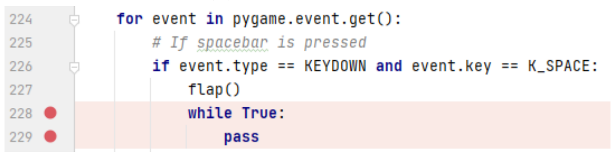

```python
    # Bird dynamics - velocity along the y axis
    self.velocity_flap = -9
```
(o) original code

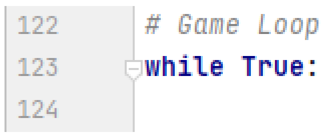

```python
    # Bird dynamics - velocity along the y axis
    self.velocity_flap = -9 * 1.25
```
(p) AVI mutant

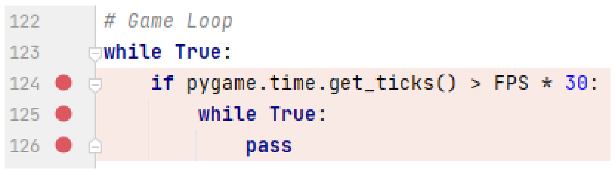

```python
    # Bird dynamics - velocity along the y axis
    self.velocity_flap = -9
```
(q) original code

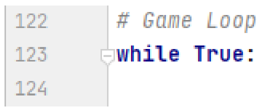

```python
    # Bird dynamics - velocity along the y axis
    self.velocity_flap = -9 * 0.25
```
(r) AVD mutant


```python
for event in pygame.event.get():
    # If spacebar is pressed
    if event.type == KEYDOWN and event.key == K_SPACE:
        flap()
```
(s) original code


```python
for event in pygame.event.get():
    # If spacebar is pressed
    if event.type == KEYDOWN and event.key == K_SPACE:
        flap()
        while True:
            pass
```
(t) GFA mutant


```python
# Game Loop
while True:
```
(u) original code


```python
# Game Loop
while True:
    if pygame.time.get_ticks() > FPS * 30:
        while True:
            pass
```
(v) GFT mutant


```python
# Game Loop
while True:
```
(w) original code

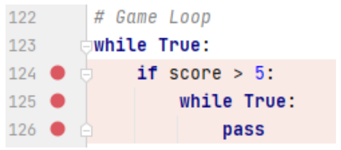

```python
# Game Loop
while True:
    if score > 5:
        while True:
            pass
```
(x) GFR mutant

Figure 1: Examples of mutation operators.

References

[1] J. A. do Prado Lima and S. R. Vergilio, "A systematic mapping study on higher order mutation testing," *Journal of Systems and Software*, vol. 154, pp. 92–109, 2019.

[2] L. Deng, J. Offutt, P. Ammann, and N. Mirzaei, "Mutation operators for testing android apps," *Information and Software Technology*, vol. 81, pp. 154–168, 2017.

[3] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "Mujava: An automated class mutation system," *Software Testing, Verification and Reliability*, vol. 15, no. 2, pp. 97–133, 2005.

[4] N. A. Butt, S. Sherin, M. U. Khan, A. A. Jilani, and M. Z. Iqbal, "Deriving and evaluating a detailed taxonomy of game bugs," 2023. arXiv: `2311.16645 [cs.SE]`.

[5] A. J. Offutt, J. Voas, and J. Payne, "Mutation operators for ada," Citeseer, Tech. Rep., 1996.

[6] M. W. Dye, C. S. Green, and D. Bavelier, "Increasing speed of processing with action video games," *Current directions in psychological science*, vol. 18, no. 6, pp. 321–326, 2009.