Implementazione digitale di giochi da tavolo

Fuzzing della GUI di Havannah

Relazione di progetto con specifiche, documentazione sperimentale e risultati finali

Campo	**Valore**
Studente	Jad Taljabini
Progetto	Implementazione digitale di Havannah in Rulebook e fuzzing della GUI
Tecnologie principali	Rulebook/RLC, Python, Tkinter, fuzzing coverage-guided, oracle a invarianti

Abstract

Questo lavoro presenta la progettazione, implementazione e valutazione sperimentale di un fuzzer per la GUI Python di Havannah. Il gioco è implementato in Rulebook, compilato tramite RLC e collegato a una GUI Tkinter tramite wrapper Python. Il fuzzer genera sequenze di eventi GUI, osserva lo stato prima e dopo ogni evento, costruisce una coverage semantica basata sullo stato del gioco e dell'interfaccia, rileva bug tramite un oracle basato su invarianti, cioè un insieme di proprietà attese sul comportamento osservabile della GUI.

A partire da una ricerca bibliografica sullo stato dell'arte del fuzz testing, il progetto si confronta con due riferimenti principali: Klees et al., usato per impostare una valutazione sperimentale più corretta, e Woo et al., usato come riferimento per lo scheduling black-box. Il contributo del lavoro non è una replica diretta dei paper, ma un adattamento delle loro idee a un contesto diverso: una GUI action-based, dove gli input non sono file mutati ma sequenze di eventi come click, reset, redraw e aggiornamenti di status.

Per rendere il confronto controllabile, è stato costruito un benchmark sperimentale basato su una GUI difettosa ma nota. Ogni violazione dell'oracle produce un bug base, mentre le varianti sintetiche rappresentano più manifestazioni contestuali dello stesso bug base. Il parametro X, chiamato bug-variant-mod, controlla quante varianti sintetiche possono essere generate per ciascuna famiglia di bug tramite un hash stabile del contesto della failure. La campagna finale confronta strategie random, coverage-guided, rate-based, Rich-Get-Richer e Anti-Rich-Get-Richer. Le metriche principali sono bug sintetici unici per run, bug sintetici unici al secondo, bug per sample, sample al secondo e coverage semantica. In questo contesto, un sample corrisponde a una singola esecuzione di una sequenza di eventi GUI.

I risultati mostrano che la differenza tra strategie diventa significativa quando aumenta il numero di varianti sintetiche associate a ciascun bug base. Nel caso più ricco, X=500, la strategia Rich-Get-Richer normalizzata per tempo, implementata come rgr_positive_rate, ottiene il miglior risultato medio in bug sintetici unici per run e in bug sintetici unici al secondo. Questa strategia premia le entry del corpus che hanno già trovato bug, ma normalizza il reward rispetto al tempo osservato. Il risultato indica che, nel contesto GUI considerato, una politica che ritorna verso traiettorie già produttive può migliorare il bug finding rate quando i bug sono distribuiti in famiglie clusterizzate. Il confronto con bug per sample mostra però che questo vantaggio dipende anche dal costo temporale delle sequenze generate, quindi non implica una superiorità universale della strategia.

La conclusione è che lo scheduler va scelto in base alla distribuzione dei bug e al costo temporale delle configurazioni esplorate. Inoltre, il lavoro analizza anche se l'ultima azione GUI eseguita prima di una failure può aiutare a raggruppare errori simili. Per verificarlo, le failure sono state divise in gruppi in base a last_action e, per ogni gruppo, è stato misurato quanto spesso le failure appartengono allo stesso bug base. Questa misura, chiamata purity, vale 99.05% su tutti gli hit e 95.60% considerando solo la prima occorrenza di ciascun bug sintetico unico per run. Il risultato suggerisce che, in sistemi GUI/action-based, l'ultima transizione osservata può essere una proxy utile per una prima deduplicazione delle failure, pur senza sostituire l'oracle come ground truth.

Mappatura rispetto ai criteri di consegna

Criterio	**Dove si trova nel documento**
Introduzione, titolo, autori, durata, contenuti	Sezioni 1 e 2
Lavori correlati e confronto comparativo	Sezione 3
Innovazione del progetto	Sezione 4
Sviluppo del progetto, strumenti, piattaforme	Sezione 5
Documenti di specifica: BPMN, schemi, UML, UX	Sezione 6
Risultati e grafici	Sezione 7
Conclusioni	Sezione 8
Appendici tecniche e riferimenti bibliografici	Appendici A e B

Indice dei contenuti

1. Introduzione generale

Havannah è un gioco da tavolo astratto su griglia esagonale in cui i giocatori alternano mosse cercando di realizzare una delle strutture vincenti: bridge, fork o ring. Nel progetto il motore del gioco è stato implementato in Rulebook, mentre l'interfaccia grafica è stata sviluppata in Python con Tkinter. Questa scelta separa chiaramente il modello logico, cioè le regole di gioco, dalla parte di presentazione e interazione utente.

La separazione tra motore e GUI è utile, ma introduce un problema di testing: verificare solo il motore non basta. Una GUI può essere corretta dal punto di vista del modello ma sbagliare il rendering, non aggiornare uno status testuale, gestire male un reset o reagire in modo incoerente a un click non valido. Per questo il progetto ha esteso l'attività di fuzzing dal semplice modello di gioco all'interfaccia grafica.

Il fuzzing è una tecnica di testing automatico in cui un programma viene eseguito molte volte con input generati o mutati automaticamente, con l'obiettivo di esplorare stati difficili da raggiungere manualmente e individuare crash, errori o comportamenti incoerenti. Nei fuzzer tradizionali gli input sono spesso file o byte stream; in questo progetto, invece, l'input del fuzzer è una sequenza di eventi GUI, come click, reset, redraw e aggiornamenti dello status. Il fuzzer non verifica soltanto se il programma termina correttamente, ma confronta lo stato osservabile della GUI con un insieme di proprietà attese sul comportamento dell'interfaccia.

Il fuzzer realizzato genera sequenze di eventi GUI. Ogni sequenza può contenere click validi, click su celle occupate, redraw, status update e reset della board. Dopo ogni evento il fuzzer prende uno snapshot dello stato osservabile e lo confronta con invarianti attesi. Se un invariante viene violato, l'esecuzione viene registrata come bug.

Obiettivo del lavoro

L'obiettivo principale è studiare come diverse strategie di scelta del corpus influenzano la capacità del fuzzer di trovare bug unici nel tempo. In particolare, il lavoro confronta modalità random, coverage-guided, rate-based, Rich-Get-Richer e Anti-Rich-Get-Richer. Il punto non è dimostrare che una tecnica vince sempre, ma capire in quali condizioni una strategia diventa vantaggiosa o controproducente.

Contributi principali

- Implementazione di un fuzzer GUI action-based per Havannah.
- Definizione di coverage semantica basata sullo stato del gioco e della GUI.
- Costruzione di un oracle a invarianti per individuare incoerenze osservabili.
- Introduzione di bug sintetici configurabili tramite bug-variant-mod X.
- Confronto sperimentale tra strategie standard, rate-based, RGR e Anti-RGR.
- Analisi del trade-off tra sample per secondo, bug per sample e bug per secondo.

2. Contesto del progetto e sistema sotto test

Il sistema sotto test è formato da tre livelli: il motore Rulebook, il wrapper Python generato da RLC e la GUI Tkinter. Il motore conserva lo stato permanente della partita: celle, turno, fine partita, vincitore, tipo di vittoria e numero di mosse. La GUI si occupa invece di costruire la finestra, disegnare la board, gestire click e reset e aggiornare le label testuali.

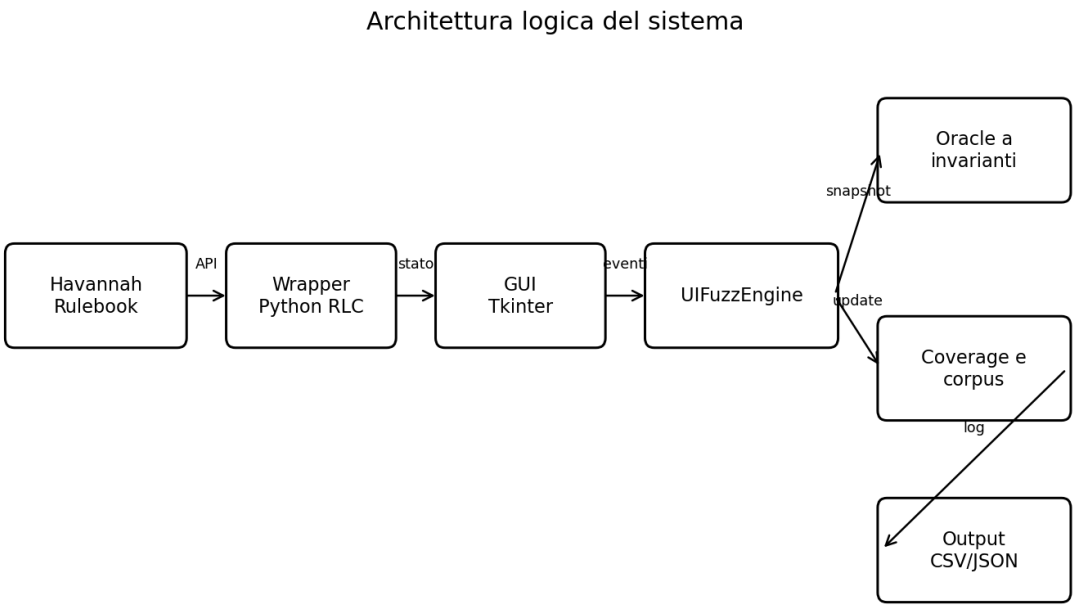

Figura 1. Architettura logica del sistema sotto test.

Dal punto di vista del testing, questo sistema è interessante perché contiene sia logica di dominio sia comportamento interattivo. Il modello può essere testato con sequenze di mosse valide, ma la GUI richiede eventi più vari: click su celle occupate, update dello status, redraw e reset. Questi eventi non sono errori di per sé, ma possono esporre incoerenze se il rendering o lo stato testuale non rimangono allineati al modello.

Target reale e target controllato

Sono stati distinti due target. Il primo è gui.py, cioè la GUI reale. Il secondo è gui_buggy.py, una versione intenzionalmente difettosa ma controllata. La GUI buggy non contiene semplici crash artificiali: contiene comportamenti sbagliati osservabili, per esempio canvas non pulito dopo un reset, pedina non disegnata dopo un redraw, label della board rimasta vecchia o status terminale incoerente.

La GUI reale è stata usata come controllo qualitativo. La GUI buggy è stata usata come benchmark sperimentale, perché permette di misurare se il fuzzer riesce a trovare famiglie di difetti note e confrontare le strategie in modo riproducibile.

3. Related work

La parte teorica del progetto si appoggia a due lavori principali: Evaluating Fuzz Testing di Klees et al. (https://cseweb.ucsd.edu/~dstefan/cse227-spring20/papers/klees:evaluating.pdf) e Scheduling Black-box Mutational Fuzzing di Woo, Cha, Gottlieb e Brumley (https://dl.acm.org/doi/epdf/10.1145/2508859.2516736). I due lavori hanno ruoli diversi. Il primo fornisce criteri metodologici per valutare un fuzzer. Il secondo fornisce l'idea di scheduling delle configurazioni e introduce metriche come Rich Gets Richer e Rate.

3.1 Evaluating Fuzz Testing

Klees et al. mostrano che valutare un fuzzer è delicato perché il fuzzing è casuale. Una singola run può portare a conclusioni sbagliate, così come una scelta arbitraria dei seed o un timeout troppo breve. Il paper critica anche l'uso ingenuo della coverage come unico indicatore: maggiore coverage può essere utile, ma non coincide automaticamente con maggior numero di bug reali trovati.

Nel progetto questi principi sono stati applicati in forma ridotta ma concreta. La campagna finale usa tre seed per ogni configurazione, un budget temporale costante, una baseline random e metriche separate: bug unici, bug per secondo, sample per secondo, bug per sample, coverage e distribuzione delle board.

3.2 Scheduling Black-box Mutational Fuzzing

Il paper sul black-box mutational fuzzing affronta il problema di scegliere quale configurazione fuzzare in una campagna con più coppie programma-seed. L'obiettivo è massimizzare il numero di bug unici scoperti nel tempo. La scelta tra esplorazione e sfruttamento viene modellata come problema di scheduling online.

La metrica Rich Gets Richer parte dall'idea che una configurazione che ha già trovato bug possa trovarsi in una zona bug-ridden e quindi meritare più budget. La metrica Rate introduce una normalizzazione rispetto al tempo, per evitare di premiare configurazioni lente solo perché producono risultati assoluti maggiori.

3.3 Differenza rispetto al contesto originale

Il progetto non replica esattamente il paper black-box. Nel paper le configurazioni sono tipicamente coppie programma-seed. Qui invece lo scheduling avviene dentro il corpus di un singolo fuzzer GUI, dove le entry sono sequenze di eventi. Questa differenza è decisiva: una metrica valida per assegnare tempo tra programmi diversi può comportarsi diversamente quando viene usata per scegliere sequenze GUI interne a un corpus.

Tabella 1. Confronto sintetico dei riferimenti teorici.

Fonte	Ruolo	Uso nel progetto
Klees et al.	Valutazione sperimentale	Seed multipli, timeout, baseline, ground truth, cautela sulla coverage
Woo et al.	Scheduling black-box	RGR, Rate, trade-off esplorazione/ sfruttamento
Questo progetto	Fuzzing GUI state-based	Coverage semantica, oracle GUI, varianti sintetiche controllate

4. Innovazione del progetto

L'elemento innovativo del progetto non è semplicemente aver scritto un fuzzer, ma aver adattato idee di fuzzing e scheduling a un dominio action-based con stato semantico confrontabile. In molti fuzzer classici la coverage è legata all'esecuzione del codice, per esempio rami o edge di controllo. In questo lavoro, invece, la coverage descrive stati significativi della partita e della GUI.

Il secondo elemento innovativo è il benchmark parametrico con bug sintetici. Ogni violazione dell'oracle produce un bug base. Con bug-variant-mod X, lo stesso bug base può generare più varianti sintetiche in base al contesto: ultima azione, board size, profondità, tipo di evento, coda della sequenza e stato terminale. In questo modo X controlla quanto una famiglia di bug è ricca di varianti, senza scrivere centinaia di bug artificiali a mano.

Questa scelta permette di studiare una domanda sperimentale precisa: quando conviene premiare una traiettoria che ha già trovato bug e quando conviene invece penalizzarla per esplorare altrove? La risposta dipende dalla distribuzione dei bug. Se i bug sono pochi e saturabili, tutte le strategie arrivano velocemente al massimo. Se una famiglia di bug produce molte varianti, allora le strategie Rich-Get-Richer diventano vantaggiose.

5. Sviluppo del progetto

Lo sviluppo del progetto si è articolato in più fasi successive. All'inizio è stato scelto Havannah come gioco da implementare in Rulebook, con l'idea di costruire prima il motore logico, poi una GUI giocabile e infine un fuzz engine capace di testare l'interazione grafica. Il progetto non è quindi nato direttamente come esperimento sullo scheduling, ma è cresciuto in modo progressivo: prima come implementazione del gioco, poi come applicazione interattiva e infine come piattaforma sperimentale per confrontare strategie di fuzzing.

Questa impostazione incrementale è stata utile perché ogni fase ha fornito la base per quella successiva. Il motore Rulebook ha dato uno stato formale del gioco; la GUI ha introdotto uno stato osservabile, fatto di canvas, label e status testuale; il fuzzer ha messo in relazione questi due livelli, confrontando ciò che il modello considera corretto con ciò che la GUI mostra dopo ogni evento. Il lavoro finale è quindi il risultato di più cicli di implementazione, verifica, correzione e valutazione sperimentale.

5.1 Strumenti usati

La tabella seguente riassume gli strumenti principali usati durante lo sviluppo. Rulebook e RLC sono stati usati per descrivere e compilare il motore del gioco, mentre Python è stato usato sia per la GUI sia per il fuzz engine. Gli output sperimentali sono stati salvati in JSON e CSV per rendere più semplice l'analisi successiva e la generazione dei grafici.

Tabella 2. Strumenti principali.

Strumento	**Ruolo**
Rulebook/RLC	Implementazione e compilazione del motore di Havannah; generazione del wrapper Python.
Python 3	Implementazione della GUI, del fuzzer, degli script di analisi e del parsing dei risultati.
Tkinter	Interfaccia grafica desktop per giocare ad Havannah e per esporre eventi controllabili al fuzzer.
JSON/CSV	Output sperimentali: statistiche, timeline, corpus, coverage e bug trovati.
GitHub Rulebook	Riferimento per confrontare struttura, stile e modalità di utilizzo di altri progetti Rulebook.

5.2 Scelta del gioco e impostazione iniziale

La prima scelta progettuale è stata concentrarsi su un solo gioco, Havannah, invece di realizzare più giochi superficiali. Questa scelta ha permesso di dedicare attenzione sia alla correttezza del modello sia alla qualità dell'interfaccia e del testing. Havannah è adatto al progetto perché ha una board esagonale, regole semplici da usare ma non banali da implementare, e tre condizioni di vittoria diverse: bridge, fork e ring.

Il lavoro è partito dal motore, perché senza uno stato formale corretto non sarebbe stato possibile costruire né una GUI affidabile né un oracle di fuzzing. In questa fase è stato necessario capire come rappresentare una board esagonale in modo compatibile con Rulebook e come rendere leggibili dall'esterno le informazioni importanti della partita, come celle, turno, vincitore e tipo di vittoria.

5.3 Implementazione del motore Havannah in Rulebook

Il motore del gioco è stato implementato in Rulebook. La board è rappresentata come una struttura esagonale di dimensione configurabile da 3 a 8. Ogni cella può essere vuota, occupata dal bianco oppure occupata dal nero. Per gestire la geometria della board sono state usate coordinate assiali, associate a ogni indice lineare della board. Inoltre, per ogni cella sono stati precomputati i sei vicini, così da rendere più semplice e più efficiente l'esplorazione delle componenti connesse.

Lo stato permanente del gioco contiene le celle, il turno corrente, il flag di partita conclusa, il vincitore, il tipo di vittoria e il numero di mosse giocate. Questa organizzazione ha permesso di separare chiaramente lo stato del modello dalla sua rappresentazione grafica. Il motore espone anche metodi di lettura sicuri, come get_cell, current_player, is_done_game, winner_player, win_structure e move_count. Questi metodi sono stati poi usati sia dalla GUI sia dal fuzzer per osservare lo stato della partita.

Una parte rilevante dell'implementazione riguarda il riconoscimento della vittoria. Havannah non ha una sola condizione terminale: un giocatore può vincere con un bridge, collegando due corner; con un fork, collegando tre lati; oppure con un ring, creando un ciclo che circonda una regione. Per bridge e fork è stata implementata una visita della componente connessa del giocatore, registrando corner e lati raggiunti. Per il ring sono stati usati controlli di connessione e visite direzionali, in modo da distinguere un semplice gruppo connesso da una struttura ciclica.

Dopo aver completato questa prima versione del motore, il modello è stato esteso con le funzioni specifiche richieste da Rulebook per il fuzzing e per l'interazione automatica con il gioco. Questo ha portato alla fase successiva: rendere il modello utilizzabile non solo come gioco, ma anche come target per strumenti automatici.

5.4 Integrazione con Rulebook/RLC e primitive per fuzzing

Il modello è stato quindi esteso con le primitive necessarie per l'infrastruttura Rulebook. In particolare, è stata definita una sessione di gioco che permette di scegliere la dimensione della board e poi applicare mosse finché la partita non termina. La scelta della board size è vincolata all'intervallo 3-8, mentre le mosse sono vincolate alle celle valide e libere.

Questa fase è stata importante perché Rulebook richiede di esporre le azioni in modo controllato. Il fuzzer deve sapere quali azioni sono applicabili e deve poterle eseguire senza rompere le precondizioni del modello. Per questo il codice non si limita a una funzione play_move, ma espone un flusso di gioco in cui la dimensione viene scelta all'inizio e ogni mossa successiva viene verificata prima dell'applicazione.

È stata poi implementata una funzione di fuzzing interna al modello, che prende una sequenza di byte, sceglie una board size e prova a convertirli in mosse. Questo primo livello di fuzzing serve soprattutto per testare il modello Rulebook. In seguito, il progetto è stato esteso oltre il modello, arrivando al fuzzing della GUI, che richiede eventi più ricchi rispetto alle sole mosse valide.

5.5 Sviluppo della GUI Tkinter

Una volta completato e verificato il motore, il progetto è passato alla parte grafica. La GUI è stata sviluppata in Python con Tkinter. All'avvio, l'applicazione chiede la dimensione della board; dopo la selezione, viene creata una nuova sessione Rulebook e viene disegnata la board esagonale sul canvas.

La GUI permette a due giocatori di giocare sullo stesso dispositivo. Ogni cella della board è rappresentata da un poligono esagonale. Quando l'utente clicca su una cella valida, la GUI invia la mossa al modello tramite il wrapper Python generato da RLC, aggiorna le pedine disegnate e mostra lo status della partita. Se il modello rileva una vittoria, la GUI annuncia il vincitore e il tipo di struttura, cioè bridge, fork o ring.

La separazione tra GUI e modello è stata mantenuta esplicitamente. Il modello Rulebook conserva la verità formale sulla partita, mentre la GUI rappresenta questa verità sullo schermo. Questa distinzione è fondamentale per il fuzzing: un bug GUI non deve necessariamente essere un errore del modello. Può essere una label non aggiornata, una pedina disegnata male, uno status incoerente o un canvas non pulito dopo un reset.

Dopo la realizzazione della GUI, è stato verificato che l'interfaccia permettesse effettivamente di giocare, scegliere la board, alternare i turni e rilevare la vittoria. A quel punto il progetto è passato alla fase sperimentale principale: la costruzione del fuzz engine.

5.6 Adattamento della GUI al fuzzing

Per rendere la GUI testabile automaticamente, non era sufficiente avere una finestra giocabile. Il fuzzer non può interagire con popup manuali o attendere input utente reali. Per questo sono stati aggiunti metodi specifici, come fuzz_reset_to_size e fuzz_click_cell. Questi metodi attraversano gli stessi handler della GUI, ma permettono di controllare il reset e il click direttamente da codice.

È stata inoltre prevista una modalità di test, in cui la finestra può essere nascosta e la GUI può essere aggiornata. In questo modo il fuzzer può creare una GUI, inviare eventi, osservare lo stato e proseguire automaticamente. Questa scelta mantiene il target realistico, perché gli eventi passano comunque dalla logica della GUI, ma rende possibile eseguire migliaia di test senza intervento manuale.

5.7 Prima versione del fuzz engine

Il fuzz engine è stato implementato in Python. L'idea di base è rappresentare ogni input come una sequenza di interi. Questa scelta mantiene gli input semplici da generare e mutare, ma permette di mapparli in eventi GUI diversi. Nel target GUI, una parte dell'intero determina il tipo di evento, mentre un'altra parte viene usata per scegliere la cella o la board size.

Gli eventi principali sono click_valid, click_occupied, redraw, status_update e reset_board. Il click valido simula una mossa normale; il click su cella occupata esercita la gestione di input invalido; redraw forza il ridisegno delle pedine; status_update aggiorna il testo dello status; reset_board cambia o ricrea la partita. Questa varietà è importante perché molti bug GUI non emergono giocando soltanto partite corrette. Spesso servono reset in stati profondi, click dopo la fine della partita o sequenze di eventi apparentemente secondari.

La prima versione del fuzzer includeva modalità casuali e modalità basate su corpus. random_raw genera sequenze casuali senza usare coverage o corpus. valid_random è più vicina al gioco normale, perché privilegia mosse valide. coverage_guided, invece, conserva le sequenze che producono nuova coverage e le riutilizza come base per nuove mutazioni. Questo schema riprende la struttura generale dei fuzzer coverage-guided: scegliere un input, mutarlo, eseguirlo, osservare il risultato e salvare l'input se è interessante.

5.8 Coverage semantica

Una delle scelte centrali del progetto è stata usare una coverage semantica invece di limitarsi alla coverage del codice. In un gioco da tavolo lo stato ha un significato preciso: dimensione della board, turno, numero di mosse, celle occupate, stato terminale, vincitore e struttura di vittoria. Il fuzzer può quindi osservare direttamente stati di gioco diversi, non solo rami di codice diversi.

La coverage del modello include chiavi come profondità della partita, turno corrente, conteggi delle pedine, maschere di edge e corner, regione dell'ultima mossa, contesto locale dei vicini e stato terminale. Per il target GUI sono state aggiunte anche chiavi legate all'interfaccia, come tipo di evento, stato testuale, label della board, numero di celle disegnate, numero di pedine nel canvas e conteggi di reset, redraw, click validi e click occupati.

Questa coverage è più domain-aware rispetto a una coverage generica. Il vantaggio è che permette di guidare l'esplorazione verso stati del gioco significativi. Il limite è che la coverage deve essere progettata manualmente e quindi dipende dal dominio Havannah. Questa scelta è comunque coerente con l'obiettivo del progetto: testare una GUI di gioco in cui lo stato formale è leggibile e confrontabile.

5.9 Oracle a invarianti

Accanto alla coverage è stato implementato un oracle a invarianti. L'oracle non considera bug un evento semplicemente insolito. Un reset, un redraw o un click dopo la fine della partita sono eventi leciti. Diventano bug solo se producono una incoerenza tra il modello e la GUI, oppure se violano una proprietà attesa.

Per ogni evento, il fuzzer raccoglie uno snapshot prima e uno snapshot dopo. Lo snapshot contiene informazioni osservabili della GUI e del modello: board size, move count, stato terminale, vincitore, win kind, firma delle celle, numero di pedine occupate, numero di celle e pedine nel canvas, status e label della board. L'oracle confronta queste informazioni con le proprietà attese.

Gli invarianti principali controllano che il numero di celle disegnate coincida con il numero di celle del modello, che il numero di pedine nel canvas coincida con le celle occupate, che la label della board sia coerente con la board size, che un reset produca una board pulita, che un click su cella occupata non modifichi il modello e segnali un errore, e che dopo una vittoria lo status dica chiaramente che la partita è conclusa. Sono stati aggiunti anche invarianti sul modello Rulebook, per controllare valori delle celle, contatore mosse, stato terminale e transizioni attese.

5.10 Creazione della GUI buggy controllata

Per valutare il fuzzer in modo riproducibile è stata creata una versione controllata della GUI, chiamata gui_buggy.py. Questa scelta non serve a fingere di aver trovato bug reali nella GUI finale, ma a costruire un benchmark sperimentale in cui i difetti sono noti e misurabili. In questo modo è possibile confrontare strategie diverse usando una ground truth controllata.

I bug inseriti sono artificiali ma osservabili. Per esempio, dopo certe sequenze di reset il canvas può non essere pulito correttamente; in uno stato profondo un redraw può saltare una pedina; dopo un reset la label della board può rimanere vecchia; dopo la fine della partita un click può disegnare una pedina finta; un click su cella occupata può non comunicare l'errore; dopo una vittoria complessa lo status può continuare a mostrare un turno. Sono difetti GUI realistici perché riguardano rendering, status, label e gestione degli eventi.

La GUI reale, gui.py, è stata usata come controllo qualitativo. La GUI buggy è stata usata per le campagne sperimentali, perché consente di verificare se il fuzzer riesce a raggiungere le condizioni che attivano i bug e se l'oracle li classifica correttamente.

5.11 Strategie implementate

Dopo la prima versione del fuzzer, il lavoro si è concentrato sullo scheduling del corpus. Le modalità corpus-based condividono lo stesso ciclo generale: scegliere una entry, mutarla, eseguirla, osservare coverage e bug, aggiornare il corpus se l'input è interessante. Ciò che cambia è la formula con cui viene scelta la entry da mutare.

Tabella 3. Strategie di fuzzing confrontate.

Modalità	Descrizione
random_raw	Genera sequenze casuali senza usare corpus o coverage.
coverage_guided	Sceglie entry con alto gain di coverage e penalizza esecuzioni ripetute.
coverage_guided_rate	Versione rate-based: reward su coverage e bug diviso per tempo.
rgr_positive	Premia le entry che hanno già trovato bug, assumendo zone bug-rich.
rgr_positive_rate	Versione RGR normalizzata per tempo.
anti_rgr_negative	Penalizza le entry che hanno già trovato bug, per spingere esplorazione altrove.
anti_rgr_negative_rate	Versione anti-RGR normalizzata per tempo.

Le strategie RGR e Anti-RGR sono state incluse per studiare due ipotesi opposte sulla distribuzione dei bug. RGR segue l'idea che una traiettoria che ha già trovato bug possa trovarsi in una zona ricca di bug e quindi meritare più budget. Anti-RGR rappresenta invece l'ipotesi opposta: se i bug sono distribuiti in modo più indipendente, una traiettoria che ha già trovato un bug potrebbe aver già esplorato la parte più utile di quella zona e quindi può essere penalizzata per favorire l'esplorazione di altre parti dello spazio degli input.

5.12 Configurazioni sperimentali considerate

Per valutare le strategie di scheduling sono state considerate diverse configurazioni del benchmark. Una prima configurazione usa pochi bug base, quasi indipendenti tra loro. In questo caso il problema tende a saturare rapidamente: una volta trovati i bug principali, continuare a sfruttare le stesse traiettorie non porta necessariamente un vantaggio.

Per studiare anche il caso opposto, è stato introdotto il parametro `bug_variant_mod`, indicato come `X`. Questo parametro permette a un bug base di generare più varianti sintetiche in base al contesto della failure. Con `X` piccolo, ogni famiglia di bug ha poche varianti; con `X` grande, invece, una stessa famiglia può produrre molte varianti. In questo modo il benchmark permette di simulare famiglie di bug più o meno raggruppate senza dover scrivere manualmente centinaia di difetti diversi.

Questa configurazione rende possibile confrontare in modo controllato strategie diverse, come random_raw, coverage_guided, coverage_guided_rate, rgr_positive_rate e anti_rgr_negative. L'obiettivo non è mostrare che una strategia sia sempre superiore, ma capire in quali condizioni conviene sfruttare traiettorie già produttive e in quali condizioni conviene invece spingere maggiormente l'esplorazione.

5.13 Bilanciamento del benchmark e delle modalità guidate

Per rendere più equo il confronto, sono state considerate anche configurazioni volte a ridurre vantaggi strutturali non legati allo scheduling. In particolare, random_raw tendeva a generare sequenze lunghe più facilmente rispetto alle modalità corpus-based. Questo poteva favorire il raggiungimento di stati profondi non perché la strategia fosse migliore, ma perché esplorava sequenze più lunghe fin dall'inizio.

Per questo motivo, nella configurazione finale è stato usato un corpus iniziale più bilanciato, con sequenze di lunghezza 0, 5, 10, 20 e 40 per ogni board size. È stata inoltre aggiunta una mutazione a blocchi, così le modalità guidate possono raggiungere stati profondi più rapidamente. In questo modo il confronto tra strategie dipende maggiormente dalla politica di scheduling e meno dalla diversa capacità iniziale di generare sequenze lunghe.

Anche il calcolo delle varianti sintetiche è stato reso più stabile. Invece di basarsi solo su una formula semplice legata al move_count, la variante viene calcolata usando un hash SHA-256 del contesto del bug. Il contesto include informazioni come bug base, ultima azione, tipo di evento, board size, profondità e coda della sequenza. Questo rende i valori alti di X, come X=500, effettivamente utili per simulare famiglie di bug ricche di varianti.

5.14 Output sperimentali e analisi dei log

Ogni run del fuzzer produce output strutturati. stats.json contiene le metriche aggregate della run, come tempo, numero di esecuzioni, bug trovati e coverage. timeline.csv registra l'evoluzione della run nel tempo e permette di costruire grafici bug-time e coverage-time. bugs_found.json contiene gli identificatori dei bug sintetici e dei bug base. coverage.json registra le chiavi di coverage raggiunte. corpus.json e top_corpus.json descrivono il corpus finale e le entry più interessanti.

Questi output hanno permesso di analizzare il risultato da più punti di vista: bug sintetici unici, bug per secondo, sample per secondo, bug per sample, coverage semantica, distribuzione delle board, mix degli eventi GUI e associazione tra last_action e bug base. La scelta di salvare dati strutturati in JSON e CSV è stata essenziale perché i grafici della relazione non sono stati ricavati manualmente, ma da log prodotti dal fuzzer.

5.15 Problemi incontrati e soluzioni adottate

Durante lo sviluppo sono emersi diversi problemi. Il primo è stato distinguere tra un evento strano e un bug reale della GUI. La soluzione è stata definire bug solo come violazione di invarianti osservabili, non come semplice esecuzione di un evento insolito.

Un altro problema è stato la saturazione dei sei bug base. Con pochi bug, tutte le strategie finiscono per trovare quasi tutto e diventa difficile capire quale scheduler sia migliore. Il parametro X è stato introdotto proprio per superare questo limite, creando varianti sintetiche controllate. Infine, la dominanza iniziale di random_raw ha mostrato che il confronto tra strategie può essere falsato dalla lunghezza delle sequenze generate. Il corpus iniziale profondo e la mutazione a blocchi sono stati introdotti per rendere il confronto più bilanciato.

Il risultato finale è quindi un sistema più completo rispetto all'idea iniziale. Non si tratta solo di una GUI giocabile o di un fuzzer casuale, ma di una piattaforma sperimentale che permette di confrontare strategie di scheduling su un target GUI controllato, con coverage semantica, oracle a invarianti, bug sintetici configurabili e output analizzabili.

6. Documenti di specifica

Questa sezione raccoglie i principali documenti di specifica del progetto: processi, architettura, diagrammi UML, stato e transizioni, schema dati e aspetti di UX. I diagrammi servono a comunicare in modo chiaro la struttura del sistema, le sue componenti principali e il flusso di esecuzione, senza sostituire il codice sorgente.

6.1 Processo BPMN

Il processo principale parte dalla configurazione della campagna, sceglie una modalità di fuzzing, seleziona o genera una sequenza, esegue la GUI, osserva snapshot e invarianti e aggiorna corpus e metriche. Il processo si ripete fino al raggiungimento del timeout.

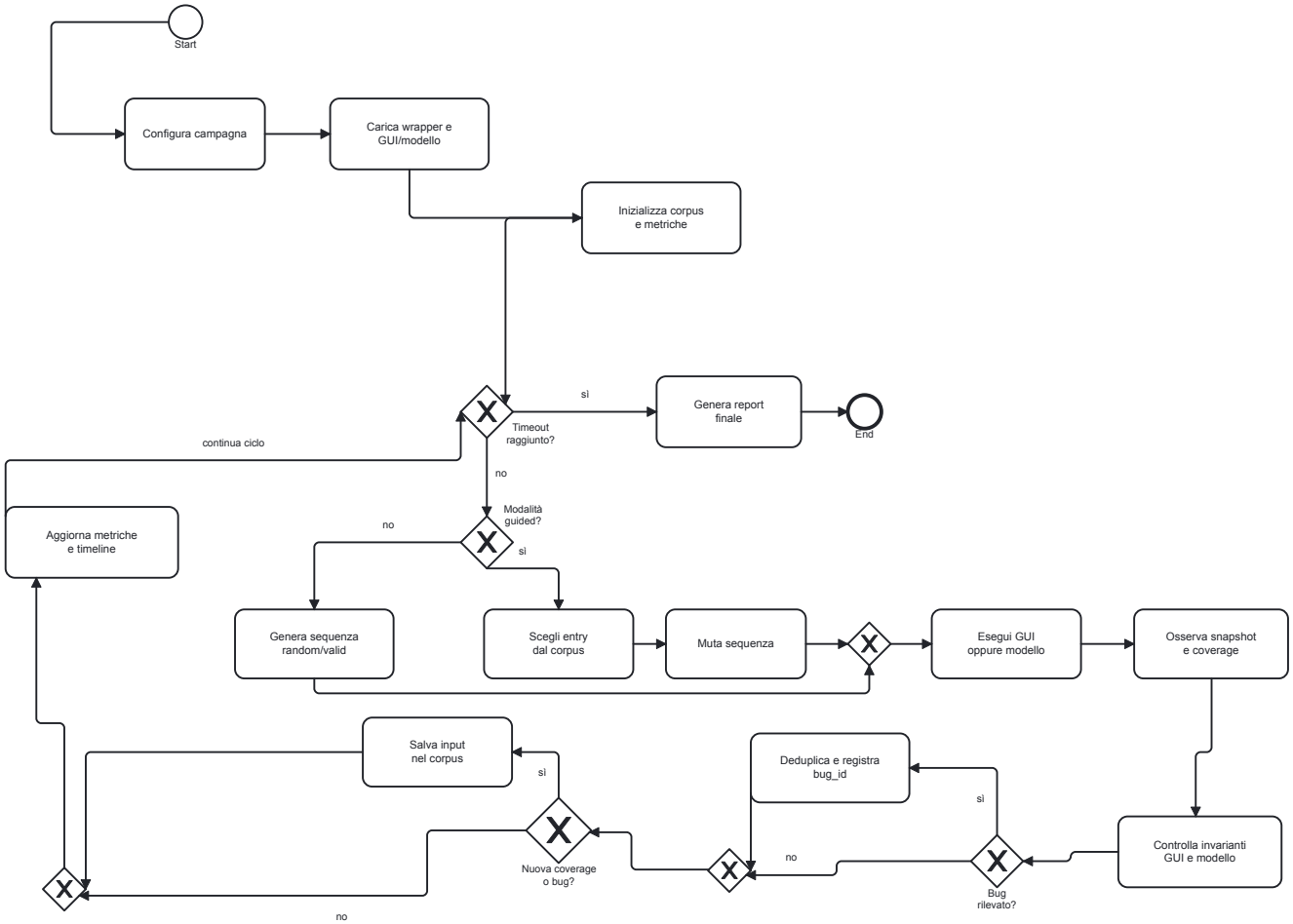

Figura 4. BPMN completo della campagna di UI fuzzing.

6.2 Block diagram architetturale

Il block diagram separa il motore Rulebook dalla GUI e dal fuzz engine. Questa separazione è importante perché consente di confrontare lo stato logico del gioco con lo stato osservabile del canvas e delle label.

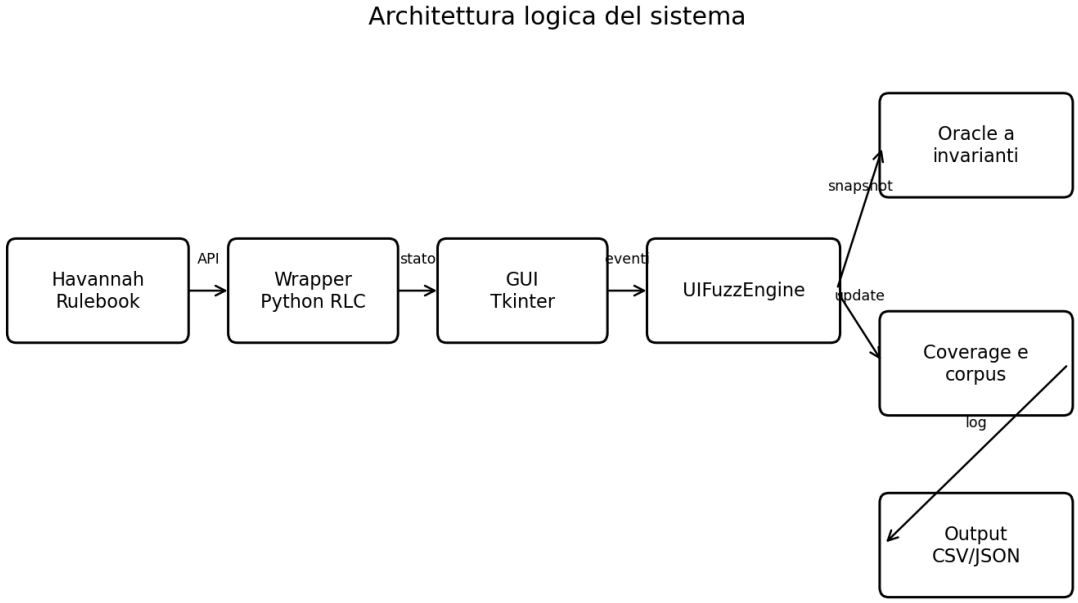

Figura 5. Block diagram del sistema.

6.3 Diagramma UML

Le classi principali del sistema sono `UIFuzzEngine`, `CorpusEntry`, `RunResult`, `HavannahGUI`, `WrapperRLC` e `HavannahGame`. `UIFuzzEngine` è il componente centrale: gestisce la campagna di fuzzing, sceglie la strategia, muta gli input, esegue le sequenze e aggiorna coverage, corpus e bug trovati.

`CorpusEntry` rappresenta una sequenza riutilizzabile dal fuzzer. Contiene la dimensione della board, la lista di mosse/eventi e alcune metriche usate per decidere se quella sequenza è interessante, come coverage prodotta, numero di esecuzioni e bug rilevati.

`RunResult` raccoglie il risultato di una singola esecuzione: input generato, mosse effettivamente applicate, coverage osservata, eventuale bug trovato, ultima azione eseguita e tempo impiegato.

`HavannahGUI` espone al fuzzer eventi controllabili, come reset della board, click su cella, redraw e aggiornamento dello status. `WrapperRLC` collega il codice Python al modello Rulebook. Infine, `HavannahGame` conserva lo stato formale del gioco, cioè celle, turno, numero di mosse, vincitore e tipo di vittoria. Il fuzzer confronta questo stato con quello osservabile nella GUI per individuare incoerenze.

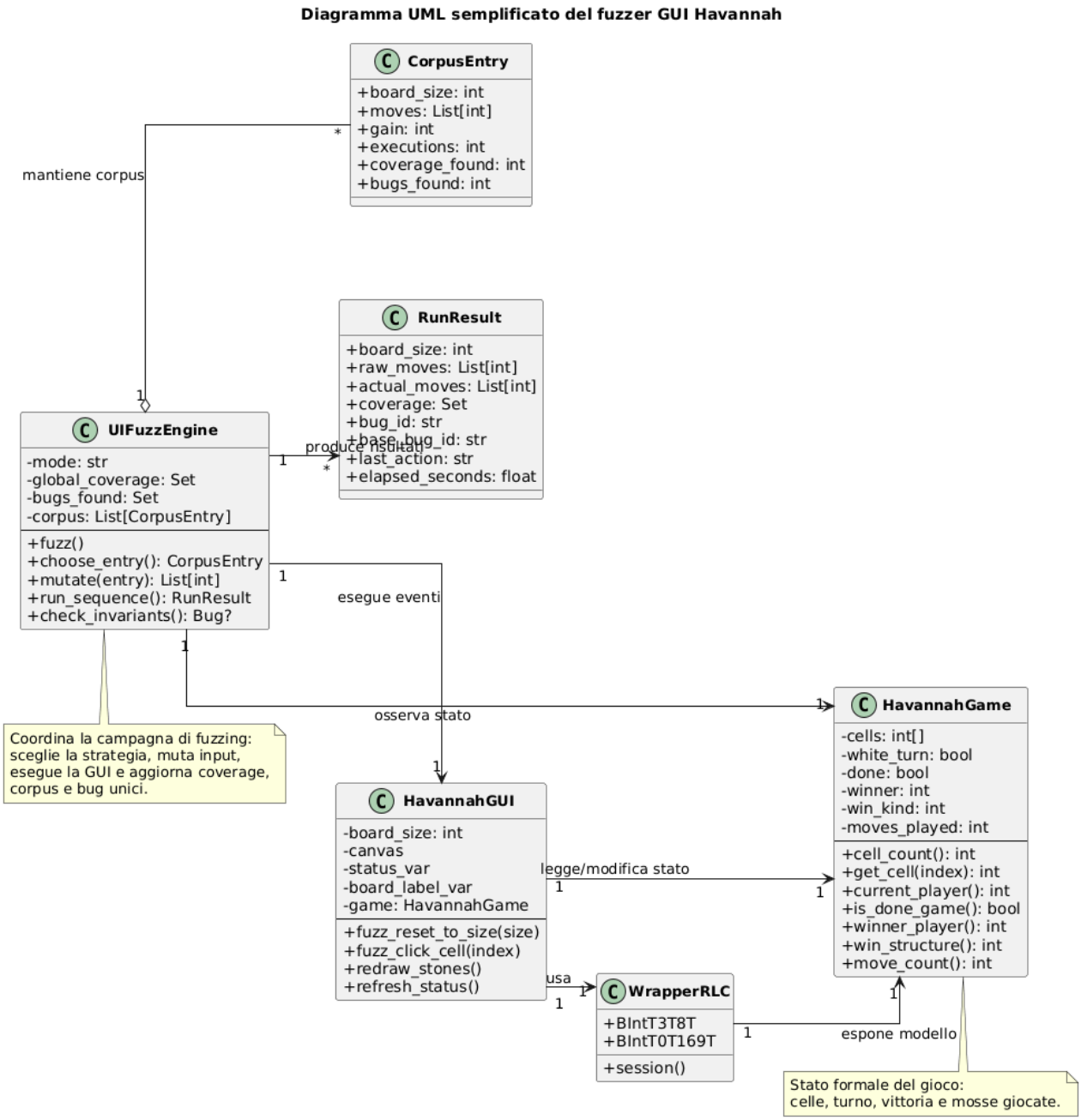

Figura 6. UML semplificato delle classi principali.

6.4 State/transition diagram

Il fuzzer attraversa stati ricorrenti: init, choose, mutate, run, oracle, update e end. Il loop centrale continua finché il budget temporale non scade. L'input entra nel corpus solo se produce coverage nuova o bug nuovo.

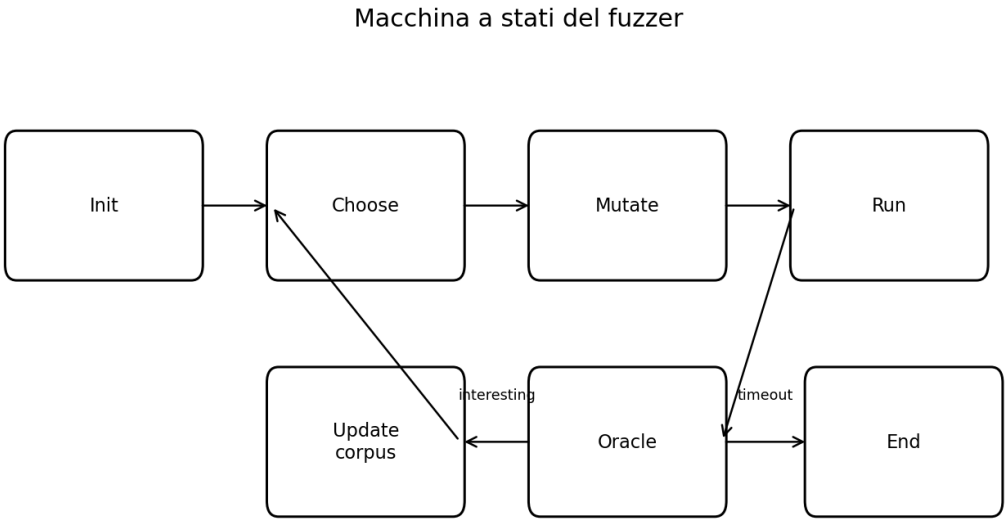

Figura 7. Diagramma di stato del fuzzer.

6.5 Sequence diagram

Il sequence diagram descrive il comportamento del fuzzer durante una singola iterazione della campagna. Prima di eseguire un evento, il fuzzer legge uno snapshot iniziale della GUI, cioè lo stato osservabile dell'interfaccia: canvas, status, label della board e informazioni principali del modello. Successivamente invia alla GUI un evento controllato, che può essere un click, un reset, un redraw o un aggiornamento dello status. La GUI, in base all'evento ricevuto, interagisce con il modello `HavannahGame`, eseguendo una mossa, resettando la partita oppure leggendo lo stato corrente. Dopo l'esecuzione dell'evento, il fuzzer raccoglie un secondo snapshot e passa all'oracle le informazioni necessarie per il controllo: stato prima, stato dopo e stato formale del modello.

L'oracle verifica gli invarianti definiti nel progetto e restituisce un esito, cioè `ok` se non sono state trovate incoerenze oppure un identificatore di bug se una proprietà attesa è stata violata. L'esito dell'iterazione viene poi salvato negli output sperimentali, in particolare nella timeline e nelle statistiche aggregate. Infine, se l'input ha prodotto nuova coverage o un nuovo bug, la sequenza viene aggiunta al corpus per essere riutilizzata e mutata nelle iterazioni successive; altrimenti viene scartata.

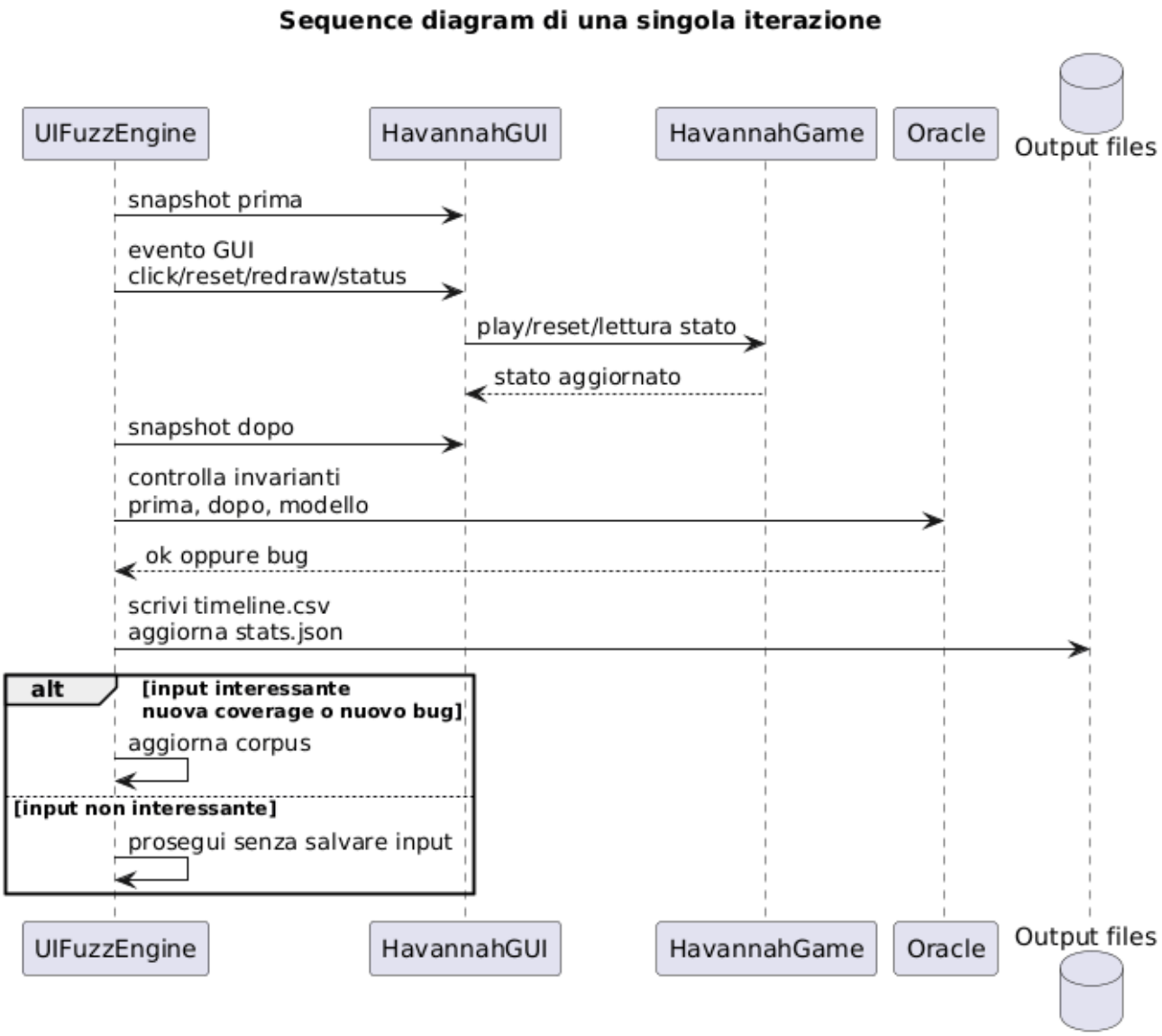

Figura 8. Sequence diagram di una singola iterazione di fuzzing.

6.6 Data schema degli output

Ogni run produce stats.json, timeline.csv, corpus.json, top_corpus.json, coverage.json e bugs_found.json. Lo schema logico collega Run, TimelineRow, Bug, CoverageKey, CorpusEntry e BoardMetric.

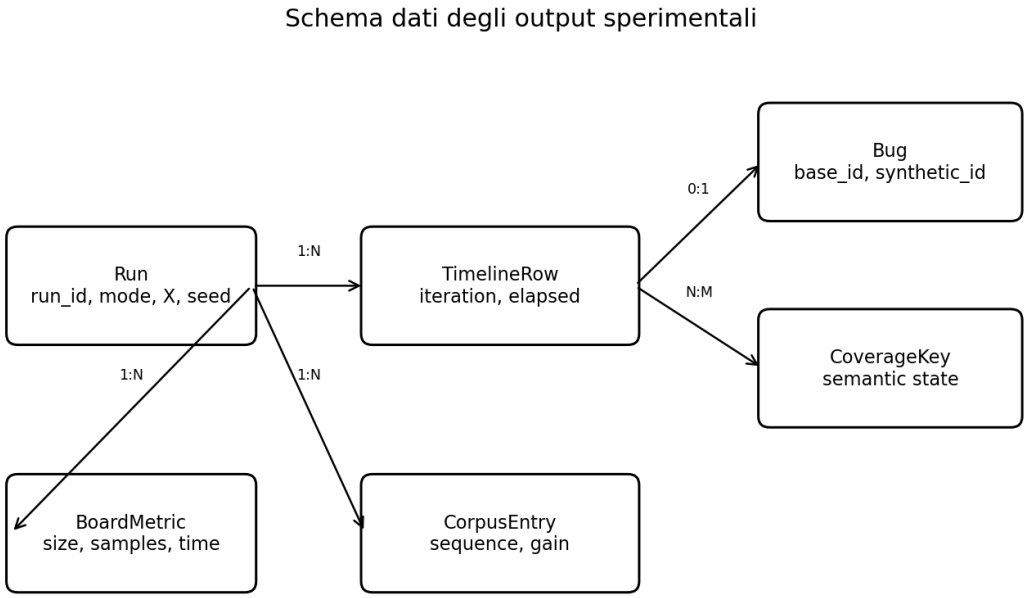

Figura 9. Schema dati degli output.

6.7 UX e manuale utente

Dal punto di vista UX, il progetto prevede due modalità di utilizzo. La prima è interattiva: l'utente avvia la GUI, sceglie la board size e gioca. La seconda è sperimentale: l'utente lancia il fuzz engine con parametri da terminale e poi analizza i file generati. La parte di fuzzing non richiede intervento manuale durante la campagna.

7. Risultati sperimentali finali

La campagna finale cluster_final_v2 è stata eseguita su gui_buggy.py. Sono stati testati quattro valori di bug-variant-mod X: 1, 10, 100 e 500. Per ogni valore sono stati usati tre seed e sette modalità di fuzzing, per un totale di 84 run. Ogni run aveva un budget nominale di 240 secondi e un numero massimo di iterazioni molto alto, così da fermarsi per timeout e non per limite di iterazioni.

Tutte le run hanno superato leggermente il budget nominale di 240 secondi per la granularità del controllo del timeout. In tre casi il tempo effettivo è risultato molto più alto del nominale; per correttezza, tutte le metriche normalizzate per tempo sono state calcolate usando elapsed_seconds reale.

Tabella 4. Setup della campagna finale.

Parametro	**Valore**
Target	gui_buggy.py
Valori X	1, 10, 100, 500
Seed	1, 2, 3
Run totali	84
Budget nominale	240 secondi per run
Eccezioni	0

Salvo dove indicato diversamente, tutti i valori riportati sono medie su tre seed. Le metriche sui bug sintetici unici sono espresse come numero medio di bug per run (bug/run). Le metriche normalizzate per tempo sono espresse in bug sintetici unici al secondo (bug/s) o sample al secondo (sample/s). Le distribuzioni delle board e degli eventi GUI sono espresse in percentuale (%). La coverage semantica è espressa come numero medio di chiavi di coverage osservate per run (keys/run).

7.1 Bug sintetici unici

Tabella 5. Bug sintetici unici medi per run su tre seed.

Modalità	**X=1 (bug/run)**	**X=10 (bug/run)**	**X=100 (bug/run)**	**X=500 (bug/run)**
random_raw	6.000	57.333	490.667	1341.667
coverage_guided	6.000	56.333	438.000	1451.667
coverage_guided_rate	6.000	59.667	518.667	1656.333
rgr_positive	6.000	58.000	487.000	1618.667
rgr_positive_rate	6.000	58.667	526.667	1811.667
anti_rgr_negative	6.000	56.333	406.000	733.667
anti_rgr_negative_rate	6.000	58.000	406.000	920.000

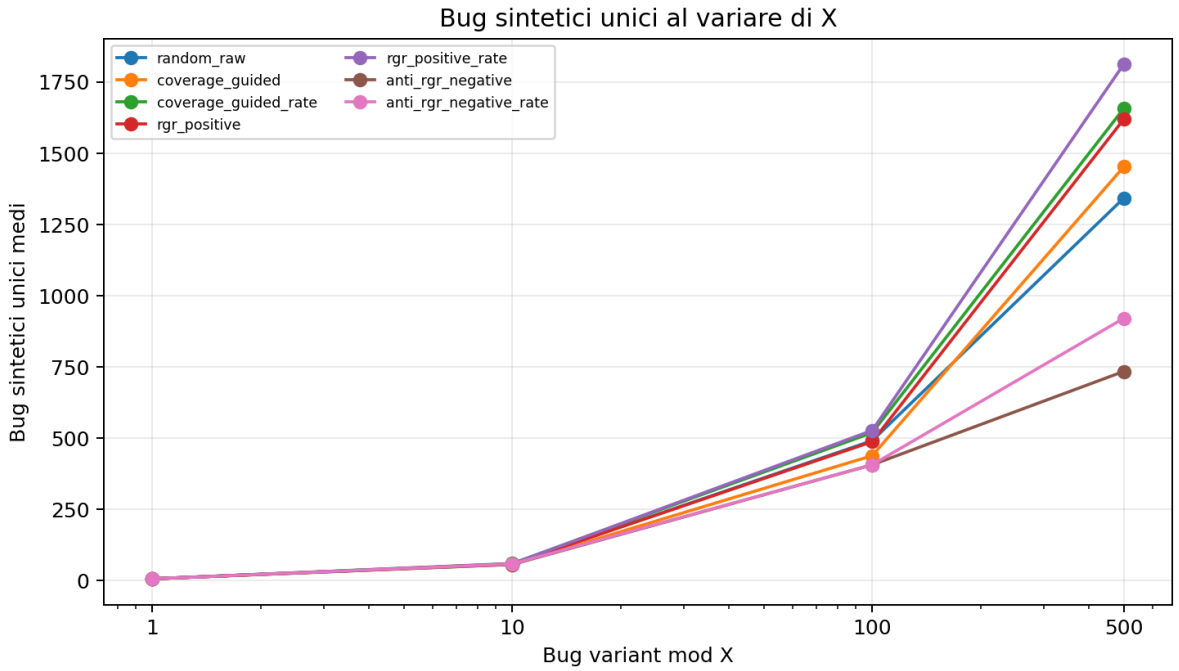

Figura 10. Bug sintetici unici medi per run al variare di X (bug/run).

Il risultato principale è chiaro. Con X=1 tutti saturano i sei bug base. Con X=10 le differenze sono minime. Con X=100 e X=500 emergono differenze più forti: rgr_positive_rate diventa la modalità migliore, seguita da coverage_guided_rate e rgr_positive. Anti-RGR resta indietro, soprattutto con X=500.

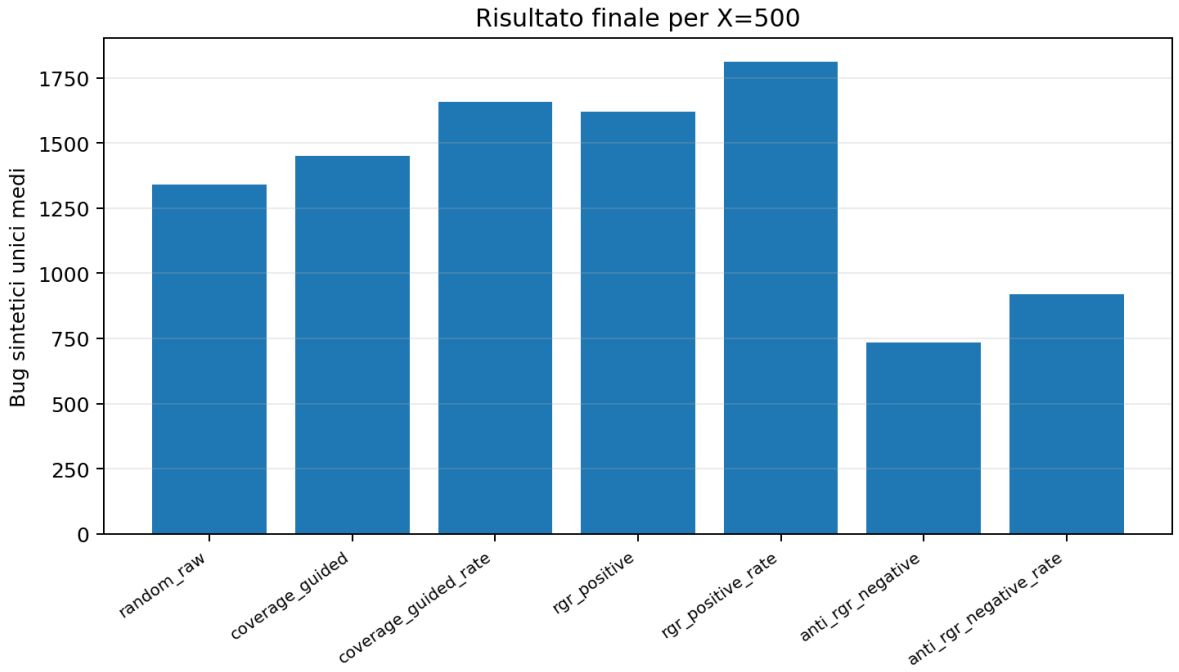

Figura 11. Confronto finale delle modalità per X=500 (bug/run).

7.2 Bug per secondo e throughput

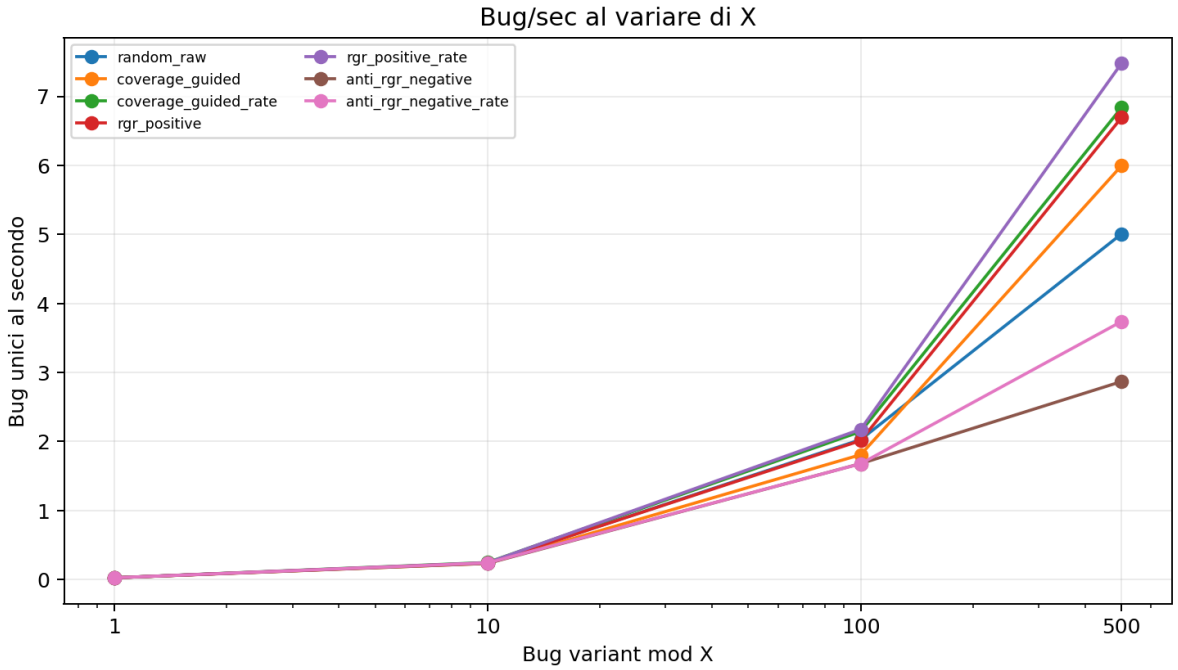

Figura 12. Bug sintetici unici al secondo al variare di X (bug/s).

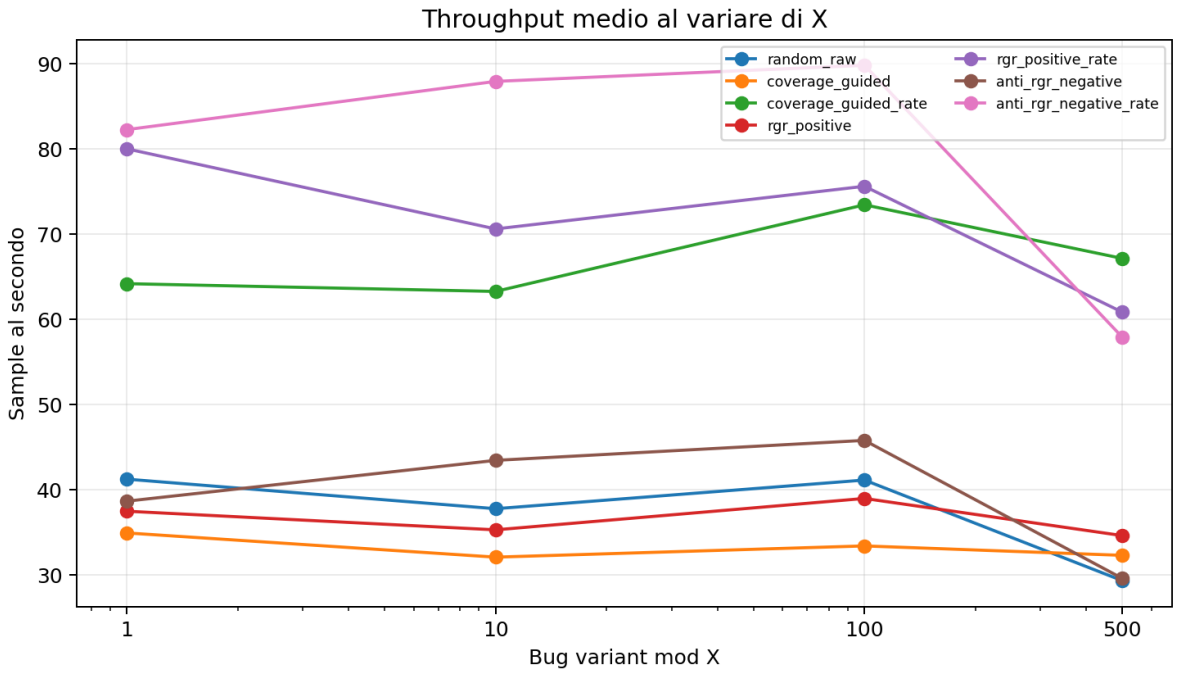

Figura 13. Throughput medio al variare di X (sample/s).

La misura bugs/sec conferma lo stesso andamento della misura assoluta dei bug unici. La normalizzazione per tempo è importante perché alcune modalità producono più sample al secondo di altre. Tuttavia, più sample al secondo non significa automaticamente più bug al secondo: conta anche la qualità delle sequenze generate.

7.3 Coverage semantica

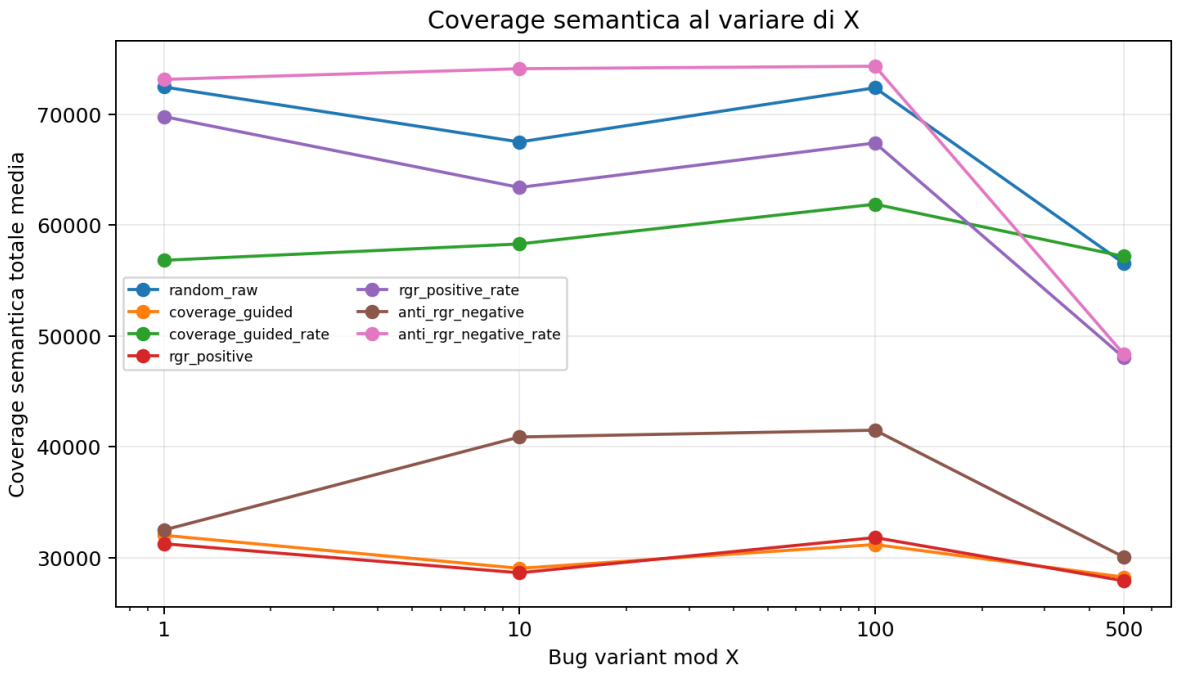

Figura 14. Coverage semantica totale media, espressa come chiavi di coverage per run (keys/run).

La coverage semantica resta una metrica utile per capire l'esplorazione, ma non sostituisce il bug finding. Alcune modalità rate ottengono throughput e coverage elevati, ma il risultato finale va interpretato insieme ai bug unici e ai bug per secondo. Questo è coerente con il messaggio metodologico di Klees et al.: coverage e bug finding sono correlate, ma non equivalenti.

7.4 Distribuzione delle board

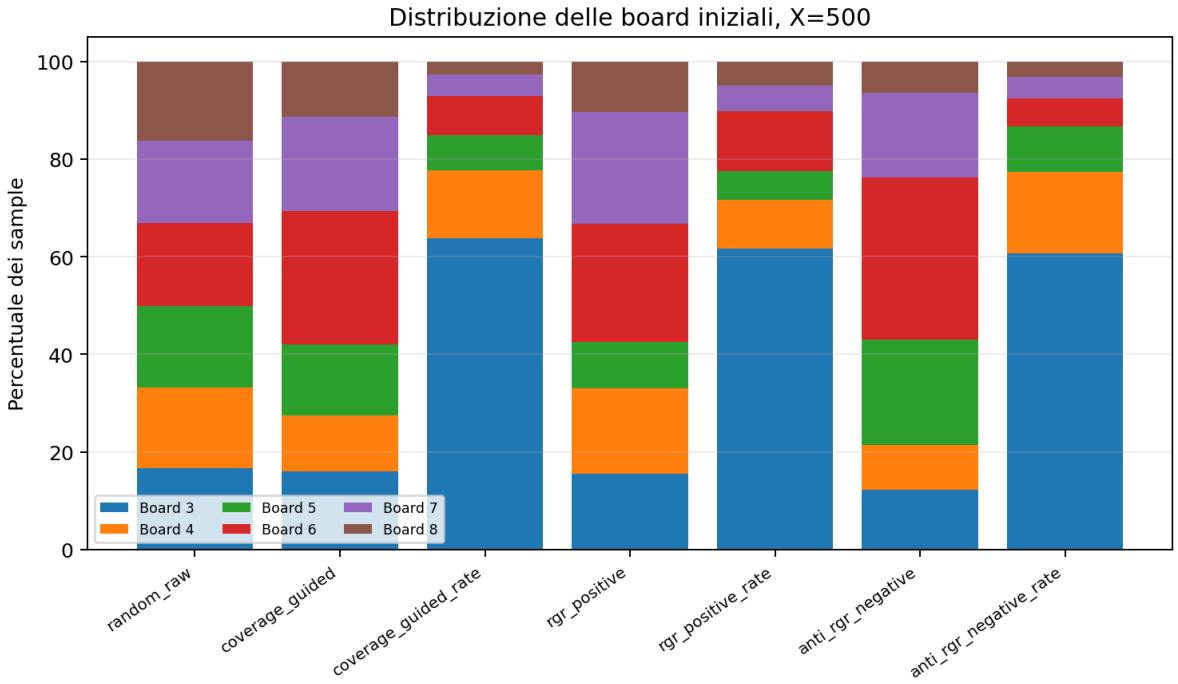

Figura 15. Distribuzione delle board iniziali per X=500, espressa come percentuale dei sample (%).

7.5 Mix degli eventi GUI

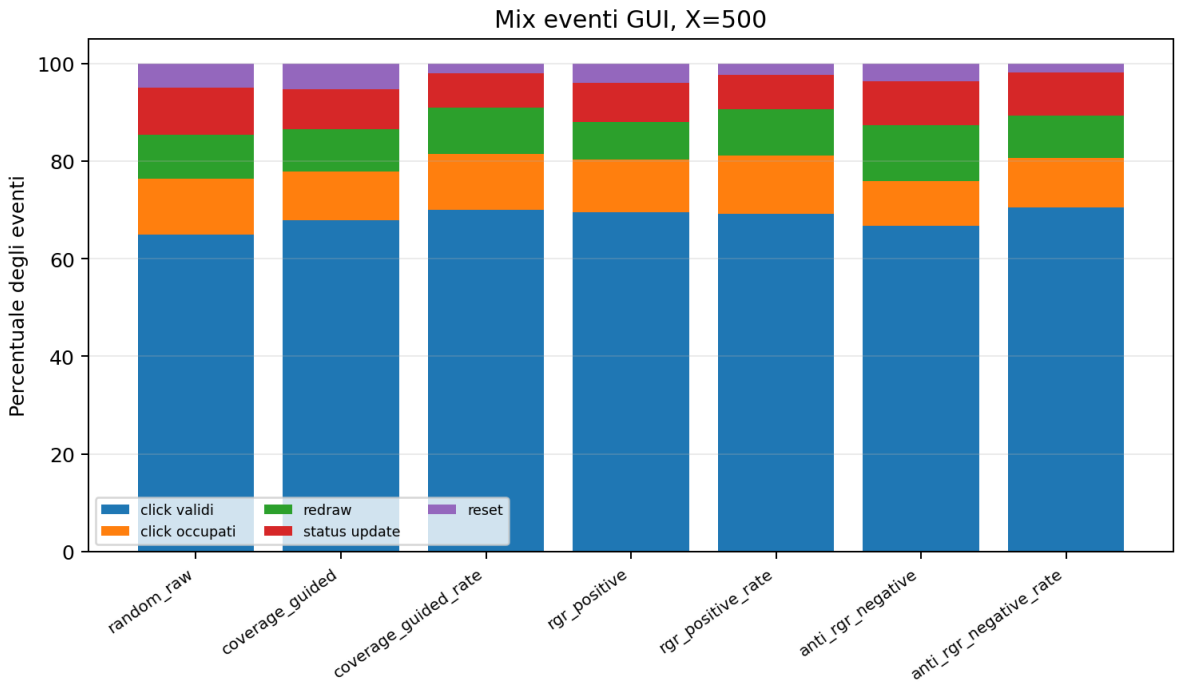

Figura 16. Mix degli eventi GUI per X=500, espresso come percentuale degli eventi eseguiti (%).

Il mix degli eventi mostra che tutte le modalità producono soprattutto click validi, ma la presenza di click occupati, redraw, status update e reset è essenziale per esercitare davvero la GUI. I bug GUI non emergono solo giocando partite valide: spesso richiedono eventi laterali o stati terminali.

7.6 Lettura del risultato rispetto alle ipotesi

L'ipotesi discussa era che il vantaggio di RGR dipenda dalla distribuzione dei bug. La campagna finale supporta questa idea. Quando X è basso, ogni famiglia contiene poche varianti e il problema satura. Quando X cresce, le famiglie di bug diventano più ricche e conviene tornare vicino alle traiettorie già produttive. In questo regime rgr_positive_rate è la modalità più forte.

La modalità anti_rgr_negative è utile come controprova. Se i bug sono clusterizzati, penalizzare le traiettorie che trovano bug è sbagliato: il fuzzer scappa proprio dalle zone più produttive. Questo spiega perché Anti-RGR crolla rispetto alle modalità RGR quando X=500.

7.7 Analisi di last_event e last_action come proxy di deduplicazione

Oltre al confronto tra strategie di scheduling, è stata analizzata anche la possibilità di usare l'ultima azione eseguita come proxy per la deduplicazione dei bug. Nei fuzzer reali, deduplicare molte failure è difficile: metriche come coverage hash o stack hash possono sovrastimare o confondere il numero di bug reali. Nel nostro caso, il target è una GUI event-driven, quindi molte failure sono legate direttamente all'evento che ha appena preceduto il fallimento.

Per ogni failure sono stati registrati last_event e last_action. last_event indica la categoria generale dell'evento, per esempio reset_board, redraw o click_after_done; last_action aggiunge anche un dettaglio contestuale, come la cella cliccata o la board scelta. Questi valori sono stati confrontati con base_bug_id, cioè il bug base identificato dall'oracle.

La metrica usata è la purity del raggruppamento. Per ogni gruppo definito dalla proxy, ad esempio tutte le failure con lo stesso last_event oppure con la stessa last_action, si individua il base_bug_id più frequente nel gruppo. La purity è il rapporto tra il numero di failure appartenenti al bug base dominante del proprio gruppo e il numero totale di failure considerate. In formula:

purity = failure assegnate al bug base dominante del gruppo / failure totali

Una purity alta significa che la proxy separa bene le failure in gruppi quasi omogenei rispetto al bug base. Non significa però che la proxy scopra il bug da sola: il riferimento rimane sempre l'oracle, usato come ground truth per calcolare la qualità del raggruppamento

I risultati mostrano che questa proxy è utile. Raggruppando tutte le failure per last_event, la purity rispetto al bug base è pari al 98.17%. Considerando invece solo la prima occorrenza di ciascun bug sintetico unico all'interno della singola run, la purity di last_event scende al 91.42%. Usando last_action, cioè una proxy più specifica perché include anche dettagli come la cella cliccata o la board scelta, la purity sale al 99.05% su tutti gli hit e al 95.60% sulle prime occorrenze dei bug sintetici unici per run.

Quindi last_action non sostituisce l'oracle e non può essere usata come identificatore certo del bug. È però una proxy leggera e promettente per raggruppare failure simili prima di una triage più precisa. Questo risultato è interessante perché sfrutta una caratteristica del dominio GUI/action-based: le failure sono spesso strettamente legate all'ultima transizione eseguita.

8. Conclusioni

Il progetto ha realizzato un fuzzer per la GUI Python di Havannah basato su sequenze di eventi, coverage semantica e oracle basato su invarianti. Rispetto a un test limitato al solo motore di gioco, il fuzzer esercita anche aspetti osservabili dell'interfaccia grafica, come rendering, status testuale, reset, redraw e gestione di input non validi. Questo permette di rilevare difetti che non appartengono direttamente alla logica Rulebook, ma alla sincronizzazione tra modello e GUI.

La campagna sperimentale finale mostra che lo scheduling del corpus non ha una strategia universalmente migliore. Quando il numero di bug distinguibili è basso, il confronto tra strategie è poco informativo, perché il benchmark raggiunge rapidamente il limite sperimentale dei bug base disponibili. Quando invece aumenta il numero di varianti sintetiche associate a ciascun bug base, le differenze diventano più significative. In particolare, nel caso X=500, la strategia Rich-Get-Richer normalizzata per tempo, implementata come rgr_positive_rate, ottiene il miglior risultato medio sia in bug sintetici unici per run sia in bug sintetici unici al secondo.

Questo risultato è coerente con l'intuizione del paper sullo scheduling black-box: se una traiettoria ha già trovato bug, può trovarsi in una zona ricca di failure correlate e quindi meritare più budget. Tuttavia, il confronto con bug per sample e throughput mostra che il vantaggio non dipende solo dalla capacità dello scheduler di scegliere input più promettenti, ma anche dal costo temporale delle sequenze generate. Le strategie normalizzate per tempo tendono infatti a privilegiare configurazioni più economiche da eseguire, come board piccole o sequenze più rapide. Per questo il risultato non va interpretato come superiorità assoluta di RGR, ma come evidenza che RGR funziona bene quando la distribuzione dei bug è effettivamente clusterizzata.

Il contributo principale è quindi metodologico. Il benchmark parametrico basato su bug-variant-mod permette di osservare il passaggio da un regime semplice, in cui pochi bug base vengono raggiunti rapidamente, a un regime più ricco, in cui una stessa famiglia di bug può produrre molte varianti sintetiche. Questo rende possibile studiare in modo controllato quando conviene sfruttare traiettorie già produttive e quando invece è necessario esplorare maggiormente il corpus.

Un secondo risultato riguarda la deduplicazione preliminare delle failure. L'analisi di last_event e last_action mostra che, in un sistema GUI/action-based, l'ultima transizione eseguita prima della failure contiene informazione utile sul bug base osservato. In particolare, last_action raggiunge una purity del 99.05% su tutti gli hit e del 95.60% sulle prime occorrenze dei bug sintetici unici per run. Questo non sostituisce l'oracle basato su invarianti, ma suggerisce che le transizioni semantiche della GUI possono aiutare a raggruppare failure simili prima di una triage più precisa.

In conclusione, il progetto mostra che le idee di valutazione sperimentale e scheduling del fuzzing possono essere adattate anche a un contesto diverso da quello classico dei file mutati: una GUI event-driven con stato semantico osservabile. I risultati ottenuti non dimostrano che una strategia sia sempre migliore delle altre, ma mostrano che la scelta dello scheduler deve dipendere dalla distribuzione dei bug, dal costo temporale delle configurazioni esplorate e dal livello di informazione semantica disponibile sul sistema sotto test.

Limiti

- Il benchmark usa bug artificiali controllati, non bug reali scoperti casualmente in produzione. Questa scelta rende possibile una valutazione riproducibile, ma limita la generalizzazione dei risultati.
- Sono stati usati tre seed per configurazione. Questo è sufficiente per una relazione descrittiva, ma non per una valutazione statistica forte come quelle raccomandate nella letteratura sul fuzz testing.
- Il target principale è una sola GUI, quindi non è corretto estendere direttamente le conclusioni a tutte le interfacce grafiche o a tutti i giochi.

Appendice A. Invarianti dell’oracle

Tabella 7. Principali invarianti usati dall’oracle.

ID	Significato
BUG_GUI_01	Canvas non pulito o celle canvas incoerenti dopo reset.
BUG_GUI_02	Numero di pedine disegnate diverso dalle celle occupate.
BUG_GUI_03	Label della board non coerente con board size reale.
BUG_GUI_04	Click dopo fine partita disegna una pedina finta.
BUG_GUI_06	Click su cella occupata non comunica correttamente errore.
BUG_GUI_09	Status terminale incoerente con vittoria o patta.
BUG_MODEL_00-13	Invarianti sul modello Rulebook e sulle transizioni attese.

Gli invarianti GUI servono per il benchmark controllato. Gli invarianti MODEL servono invece per controllare coerenza interna del modello e transizioni, e sono quelli più adatti a un controllo prudente sulla GUI reale.

Appendice B. Riferimenti bibliografici

- George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, Michael Hicks. Evaluating Fuzz Testing. ACM CCS 2018
- Maverick Woo, Sang Kil Cha, Samantha Gottlieb, David Brumley. Scheduling Black-box Mutational Fuzzing. ACM CCS 2013
- Documentazione e materiali del progetto Rulebook/RLC usati per l'implementazione del gioco e la generazione del wrapper Python.