

Improving the QA testing process of desktop, web, mobile, and XR applications through scriptless testing using the TESTAR tool



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Fernando Pastor Ricós

Department of Computer Systems and Computation
Valencian Research Institute for Artificial Intelligence

This dissertation is submitted for the PhD Program in
Computer Science

Supervisors:

Dra. Tanja Ernestina Jozefina Vos

Dra. Beatriz Mariela Marín Campusano

València, Spain

November 2024

Thanks to my supervisors, coworkers, family, and friends.

Acknowledgements

This thesis has been developed at the Universitat Politècnica de València and has been funded by various European research projects: ITEA3 TESTOMAT project (TESTOMAT 16032), H2020 DEveloper COmpanion for Documented and annotatEd code Reference (DECODER 824231), H2020 Intelligent Verification/Validation for Extended Reality Based Systems (IV4XR 856716), ITEA3 Industrial-grade Verification and Validation of Evolving Systems (IVVES 18022), Erasmus+ European Innovation Alliance for Testing Education (ENACTEST 101055874), NWO OTP project Automated Unobtrusive Techniques for LINK-ing requirements and testing in agile software development (AUTOLINK 19521).

Abstract

Software has become a central pillar in the daily lives of people worldwide, including sectors such as banking, video games, healthcare, and aviation. To ensure trust and reliability in these software systems, it is of paramount importance to develop high-quality software. Software testing is the most widely used process to assure software quality. By rigorously exercising software systems and collecting evidence, testing enables informed decisions about a software product's quality.

Human testers play a vital role in software testing because they bring intuition, creativity, and a deep understanding of real-world contexts that automated systems cannot easily replicate. However, testers' efforts should not be misdirected to execute manual and repetitive software interactions, which lead to heavy workloads, time constraints, frustration, and fatigue; resulting in a lack of attention and time to perform all necessary types of testing.

Scripted testing is an approach to automate the execution of testing by creating test scripts to execute sequential actions and apply test oracles to reduce the manual effort involved in repetitive regression testing tasks. This approach reduces testers' effort and helps to maintain the software quality by detecting unintended regression failures. However, the need to maintain these scripts and the unmanageable number of scripts required for large and complex systems make scripted testing an insufficient approach.

Scriptless testing is a complementary automated approach that does not rely on scripts but dynamically explores software with non-sequential actions. This approach introduces randomness, which helps to uncover states and failures not identified by manual or scripted testing.

TESTAR is an open-source scriptless test automation tool that has demonstrated its coverage and fault detection effectiveness when testing industrial desktop and web applications through the Graphical User Interface (GUI). The modular architecture of TESTAR enables the tool for further improvement to: (1) connect with other complex software systems, (2) incorporate additional exploratory algorithms to continue enhancing test effectiveness and efficiency, (3) integrate a diverse set of test oracles to cover a wide variety of features or discover its faults, and (4) be validated in industrial environments and well-known open-source projects.

In this thesis, we conducted several research collaborations with industrial partners and open-source projects to improve TESTAR capabilities. First, we integrated technological frameworks in TESTAR to connect with complex desktop, web, mobile, and eXtended Reality (XR) applications. Second, we extended TESTAR with exploratory algorithms to improve the test effectiveness and efficiency of scriptless action selection decisions. Third, we analyzed industrial software failures to enhance test adequacy criteria and we integrated a set of test oracles capable of automatically detecting software failures. Fourth, we evaluated these novel extensions in complex software systems, including industrial applications and well-established open-source projects.

Our results empirically demonstrate the significant improvement of TESTAR capabilities. First, TESTAR can successfully connect to and test various desktop, web, mobile, and XR game applications. Second, a state model inference processes, a distributed inference approach, grammar-based action selection rules, and an interactive algorithm for navigating game environments help significantly improve the effectiveness and efficiency of scriptless exploration. Third, the integration of online test oracles and the development of a novel delta change detection tool for offline oracles enabled TESTAR to detect functional, security, visual, and misspelling issues across a diverse range of software applications, as well as use the state models to identify and highlight GUI changes between delta versions of the same software application. Fourth, TESTAR has proved to be a practical scriptless tool for complementing manual and scripted testing approaches in industrial organizations. Finally, recognizing the importance of supporting human testers beyond the scriptless approach, we collaborated to implement a Behavior-Driven Development (BDD) approach for automating regression tests of an XR industrial game system.

These results demonstrate the capabilities and benefits of using scriptless testing with TESTAR to complement the testing processes of both industrial and open-source software projects. This thesis concludes that investing in scriptless testing can significantly improve the Quality Assurance (QA) testing process of desktop, web, mobile, and XR applications by increasing coverage and detecting unexpected failures.

Resumen

El *software* se ha convertido en un pilar fundamental en la vida cotidiana de las personas, abarcando sectores como la banca, los videojuegos, la salud, y la aviación. Para asegurar la confianza y la fiabilidad en estos sistemas *software*, es esencial desarrollar *software* de alta calidad. Las pruebas de *software* constituyen el proceso más utilizado para asegurar dicha calidad. Al someter rigurosamente los sistemas a pruebas y recopilar evidencias, este proceso permite tomar decisiones fundamentadas sobre la calidad de un producto *software*.

Los *testers* humanos tienen un papel esencial en las pruebas de *software* debido a que aportan intuición, creatividad y una comprensión profunda de los contextos del mundo real que los sistemas automatizados no pueden replicar fácilmente. No obstante, los esfuerzos de los *testers* no deben centrarse en la ejecución manual y repetitiva de interacciones con el *software*, ya que ello implica una carga de trabajo excesiva, restricciones de tiempo, frustración y fatiga, lo que genera falta de atención y limita el tiempo disponible para realizar todos los tipos de pruebas necesarios.

Scripted testing es un enfoque para automatizar la ejecución de pruebas mediante la creación de *scripts* que ejecutan acciones secuenciales y aplican oráculos de prueba con el fin de reducir el esfuerzo manual asociado a las tareas repetitivas de pruebas de regresión. Este enfoque disminuye la carga de trabajo de los *testers* y contribuye a mantener la calidad del *software*, al detectar fallos de regresión no intencionados. Sin embargo, la necesidad de mantener estos *scripts*, sumada al número incontrolable de *scripts* requeridos para sistemas grandes y complejos, hace que las pruebas basadas en *scripts* resulten insuficientes.

Scriptless testing es un enfoque automatizado complementario que no depende de *scripts*, sino que explora el *software* de manera dinámica mediante acciones no secuenciales. Este enfoque introduce aleatoriedad, lo que permite descubrir estados y fallos no identificados por pruebas manuales o con *scripts*.

TESTAR es una herramienta de automatización de pruebas sin *scripts* de código abierto que ha demostrado ser eficaz en la cobertura y detección de fallos al probar aplicaciones industriales tanto de escritorio como web, a través de la Interfaz Gráfica de Usuario (GUI). La arquitectura modular de TESTAR permite seguir mejorando la herramienta en varios aspectos: (1) facilitar la conexión con otros sistemas de *software* complejos, (2) incorporar

algoritmos exploratorios adicionales para aumentar la efectividad y eficiencia de las pruebas, (3) integrar un conjunto diverso de oráculos de prueba para cubrir una amplia variedad de características y detectar sus fallos, y (4) validar su funcionamiento en entornos industriales y en proyectos de código abierto bien establecidos.

En esta tesis, hemos realizado diversas colaboraciones de investigación con socios industriales y proyectos de código abierto para mejorar múltiples capacidades de TESTAR. En primer lugar, integramos marcos tecnológicos en TESTAR para conectarlo con aplicaciones complejas de escritorio, web, móviles y de Realidad Extendida (XR). En segundo lugar, ampliamos TESTAR con algoritmos exploratorios para mejorar la efectividad y eficiencia en la selección de acciones sin *scripts*. En tercer lugar, analizamos fallos de *software* industrial para optimizar los criterios de adecuación de las pruebas, e integramos un conjunto de oráculos de prueba capaces de detectar automáticamente fallos en el *software*. Finalmente, evaluamos estas nuevas extensiones en sistemas de *software* complejos, incluyendo aplicaciones industriales y proyectos de código abierto bien establecidos.

Nuestros resultados demuestran de manera empírica una mejora significativa en las capacidades de TESTAR. En primer lugar, TESTAR puede conectarse y probar con éxito una amplia gama de aplicaciones, incluidas de escritorio, web, móviles y de juegos XR. En segundo lugar, la implementación de procesos como la inferencia de modelos de estado, un enfoque de inferencia distribuida, reglas de selección de acciones basadas en gramática y un algoritmo interactivo para la navegación en entornos de juegos ha mejorado significativamente la efectividad y eficiencia en la exploración sin *scripts*. En tercer lugar, la integración de oráculos de prueba en línea y el desarrollo de una herramienta novedosa de detección de cambios delta para oráculos fuera de línea permitieron a TESTAR detectar problemas funcionales, de seguridad, visuales y de errores tipográficos en una amplia gama de aplicaciones de *software*, así como utilizar los modelos de estado para identificar y resaltar cambios en la GUI entre versiones delta de la misma aplicación de *software*. En cuarto lugar, TESTAR ha demostrado ser una solución sin *scripts* eficaz para complementar los enfoques de pruebas manuales y basadas en *scripts* en entornos industriales. Finalmente, reconociendo la importancia de brindar soporte a los *testers* humanos más allá del enfoque sin *scripts*, colaboramos en la implementación de un enfoque de Desarrollo Guiado por Comportamiento (BDD) para automatizar pruebas de regresión en un sistema industrial de juegos XR.

Estos resultados demuestran las capacidades y beneficios de emplear pruebas sin *scripts* mediante TESTAR para complementar los procesos de prueba en proyectos de *software* industriales y de código abierto. Esta tesis concluye que invertir en pruebas sin *scripts* puede mejorar significativamente el proceso de aseguramiento de la calidad (QA) en aplicaciones de escritorio, web, móviles y XR, al aumentar la cobertura y detectar fallos inesperados.

Resum

El programari s'ha convertit en un pilar fonamental en la vida diària de les persones a escala mundial, incloent-hi sectors com la banca, els videojocs, la salut, i l'aviació. Per a garantir la confiança i la fiabilitat en aquests sistemes de programari essencials, és imprescindible desenvolupar un programari d'alta qualitat. La prova de programari és el procés més utilitzat per assegurar la qualitat del programari. Mitjançant proves rigoroses sobre els sistemes i la recopilació d'evidències, este procés permet prendre decisions informades sobre la qualitat d'un producte de programari.

Els testers humans tenen un paper fonamental en les proves de programari perquè aporten intuïció, creativitat i una comprensió profunda dels contextos del món real que els sistemes automatitzats no poden replicar fàcilment. No obstant això, els seus esforços no haurien de destinar-se a l'execució d'interaccions manuals i repetitives amb el programari, ja que això genera càrregues de treball elevades, restriccions de temps, frustració i fatiga, cosa que comporta una falta d'atenció i de temps per a dur a terme tots els tipus de prova necessaris.

Scripted testing és un enfocament per automatitzar l'execució de proves creant scripts per executar accions seqüencials i aplicar oracles de prova per reduir l'esforç manual que comporta les tasques repetitives de proves de regressió. Este enfocament reduïx l'esforç dels *testers* i ajuda a mantindre la qualitat del programari detectant fallades de regressió no desitjades. No obstant això, la necessitat de mantindre estos *scripts* i el nombre incontrolable de *scripts* requerits per a sistemes grans i complexos fan que la prova amb *scripts* siga un enfocament insuficient.

Scriptless testing és un enfocament automatitzat complementari que no es basa en *scripts*, sinó que explora dinàmicament el programari amb accions no seqüencials. Este enfocament introduïx aleatorietat, la qual ajuda a descobrir estats i falles no identificats per les proves manuals o amb *scripts*.

TESTAR és una eina d'automatització de proves sense *scripts* de codi obert que ha demostrat la seua cobertura i efectivitat en la detecció de falles quan es prova aplicacions industrials d'escriptori i web a través de la Interfície Gràfica d'Usuari (GUI). L'arquitectura modular de TESTAR permet a l'eina de millorar encara més per a: (1) connectar-se amb altres sistemes de programari complexos, (2) incorporar algorismes exploratoris addicionals

per seguir millorant l'efectivitat i l'eficiència de les proves, (3) integrar un conjunt divers d'oracles de prova per cobrir una àmplia varietat de característiques o descobrir els seus errors, i (4) ser validada en entorns industrials i projectes de codi obert coneguts.

En esta tesi, vam dur a terme diverses col·laboracions en recerca amb socis industrials i projectes de codi obert per millorar múltiples capacitats de TESTAR. Primerament, vam integrar marcs tecnològics en TESTAR per connectar amb aplicacions de programari complexes d'escriptori, web, mòbils i Realitat Virtual (XR). En segon lloc, vam ampliar TESTAR amb algoritmes exploratoris per millorar l'efectivitat i l'eficiència de les decisions de selecció d'accions sense *scripts*. En tercer lloc, vam analitzar falles de programari industrial per millorar els criteris d'adequació de les proves i vam integrar un conjunt d'oracles de prova capaços de detectar automàticament falles de programari. En quart lloc, vam avaluar estes noves extensions en sistemes de programari complexos, incloent-hi aplicacions industrials i projectes de codi obert ben establits.

Els nostres resultats demostren empíricament la millora significativa de les capacitats de TESTAR. Primerament, TESTAR pot connectar-se amb èxit i provar diverses aplicacions de programari d'escriptori, web, mòbils i de jocs XR. En segon lloc, els processos d'inferència de models d'estat, un enfocament d'inferència distribuïda, regles de selecció d'accions basades en gramàtica i un algoritme interactiu per navegar per entorns de jocs ajuden a millorar significativament l'efectivitat i l'eficiència de l'exploració sense *scripts*. En tercer lloc, la integració d'oracles de prova en línia i el desenvolupament d'una nova eina de detecció de canvis delta per oracles fora de línia han permés a TESTAR detectar problemes funcionals, de seguretat, visuals i d'errors tipogràfics en una gamma diversa d'aplicacions de programari, així com utilitzar els models d'estat per identificar i ressaltar canvis en la GUI entre versions delta de la mateixa aplicació de programari. En quart lloc, TESTAR ha demostrat ser una eina pràctica sense *scripts* per complementar els enfocaments de prova manuals i amb *scripts* en organitzacions industrials. Finalment, reconeixent la importància de donar suport als *testers* humans més enllà de l'enfocament sense *scripts*, vam col·laborar per implementar un enfocament de Desenvolupament Dirigit per Comportament (BDD) per automatitzar proves de regressió d'un sistema de jocs XR industrial.

Estos resultats demostren les capacitats i els beneficis de l'ús de la prova sense *scripts* amb TESTAR per complementar els processos de prova tant en projectes de programari industrials com de codi obert. Esta tesi conclou que invertir en la prova sense *scripts* pot millorar significativament el procés d'assegurament de la qualitat (QA) de les aplicacions d'escriptori, web, mòbils i XR augmentant la cobertura i detectant fallades inesperades.

Table of contents

List of figures	xix
List of tables	xxv
Abbreviations and Acronyms	xxvii
1 Introduction	1
1.1 Software Quality	2
1.2 Software testing	5
1.3 Graphical User Interface testing	9
1.3.1 Manual test execution	10
1.3.2 Scripted testing approach	12
1.3.3 Scriptless testing approach	13
1.4 Scriptless testing tools	15
1.4.1 TESTAR tool	16
1.5 Research Goal	19
1.5.1 Objectives	19
1.5.2 Methodology	20
1.6 Context	21
1.7 Thesis Structure	22
2 TESTAR: Software interaction	25
2.1 Interaction with Desktop, Web, and Mobile systems	26
2.1.1 Obtaining the Graphical User Interface state	27
2.1.2 Deriving Graphical User Interface actions	31
2.1.3 Virtual environments for GUI software interaction	37
2.2 Interaction with computer 3D game systems	38
2.2.1 iv4XR framework and game plugins	39
2.2.2 Observing the game environment state	41

2.2.3	Deriving game environment actions	44
2.2.4	Navigate game environments	45
2.2.5	Virtual environments for game software interaction	46
2.3	Summary	46
3	TESTAR: Software exploration	47
3.1	State and action identification for state model inference	48
3.1.1	Main mechanism for state and action identification	49
3.1.2	State Model Inference	53
3.1.3	Abstraction strategy for state and action identification	63
3.2	Grammar Action Selection Rules for Scriptless Testing	67
3.2.1	Grammar-Based Action Selection Rules in TESTAR	68
3.2.2	Grammar-based ASR in TESTAR for form-filling	69
3.2.3	Empirical Evaluation	71
3.2.4	Results	80
3.2.5	Threats to validity	85
3.3	Scriptless exploration of game environments	86
3.3.1	TESTAR for the LabRecruits experimental 3D game	87
3.3.2	LabRecruits empirical evaluation	90
3.3.3	LabRecruits Results	92
3.4	Summary	95
4	TESTAR: Distributed exploration	97
4.1	Distributed State Model Inference with TESTAR	98
4.1.1	Shared knowledge ASM	100
4.1.2	Non-Deterministic Shared knowledge ASM	103
4.1.3	Empirical Evaluation	106
4.1.4	Results	114
4.1.5	Threats to validity	122
4.2	Summary	124
5	TESTAR: Online test oracles	125
5.1	Online Graphical User Interface test oracles	127
5.1.1	Suspicious messages regular expressions	127
5.1.2	Graphical User Interface programmatic test oracles	130
5.1.3	Security web test oracles	135
5.1.4	Evaluation of online GUI oracles	138

5.2	Online Game test oracles	139
5.2.1	Programmatic test oracles for functional game aspects	139
5.2.2	Evaluation of online game test oracles	141
5.3	Test results	142
5.4	Summary	142
6	TESTAR: Offline delta test oracles	145
6.1	Systematic mapping of the literature	147
6.1.1	Methodology	147
6.1.2	Data collection	150
6.1.3	Data results	151
6.1.4	Other interesting related work topics	155
6.1.5	Actionable insights obtained from the systematic mapping of the literature	157
6.2	Delta GUI change detection approach	158
6.2.1	State model inference strategy	159
6.2.2	GUI Change Detection tool	161
6.2.3	Empirical Study	166
6.2.4	Results	175
6.2.5	Threats to validity	187
6.3	Summary	189
7	TESTAR: Scriptless testing in practice	191
7.1	Ponsse industrial experience in 2019	192
7.1.1	Ponsse Opti4G desktop SUT	192
7.1.2	Ponsse study with TESTAR	192
7.1.3	Results	194
7.2	Prodevelop industrial experience in 2020	195
7.2.1	Posidonia web SUT and testing process	195
7.2.2	Prodevelop study with TESTAR	197
7.2.3	Results	201
7.3	Kuveyt Türk industrial experience in 2021	203
7.3.1	Kuveyt Türk Bank SUT and testing process	203
7.3.2	Kuveyt Türk study with TESTAR	203
7.3.3	Results	205
7.4	ING industrial experience in 2022	205
7.4.1	ING mobile SUT and testing process	206

7.4.2	ING scriptless testing study for mobile applications	206
7.4.3	Empirical evaluation	212
7.4.4	Results	214
7.5	Keen Software House & GoodAI industrial experience in 2023	217
7.5.1	Space Engineers game SUT and testing process	218
7.5.2	Space Engineers study with the iv4XR framework	221
7.5.3	Space Engineers study with the TESTAR-iv4XR agent	223
7.5.4	Empirical evaluation	227
7.5.5	Results	229
7.6	DigiOffice industrial experience in 2023	232
7.6.1	DigiOffice web SUT and testing process	232
7.6.2	DigiOffice study with TESTAR	233
7.6.3	Results	234
7.7	Summary	235
8	Beyond TESTAR: A BDD testing approach	237
8.1	Behavior-Driven-Development game testing	238
8.1.1	Space Engineers manual regression testing practices	238
8.1.2	Test Scripts for regression test automation	241
8.1.3	BDD for regression game testing automation	242
8.1.4	Industrial application of BDD test scenarios	246
8.1.5	Threats to validity	247
8.2	Summary	248
9	Discussion	251
9.1	Interaction with Graphical User Interface systems	252
9.2	Interaction with eXtended Reality game systems	256
9.3	Abstraction strategies and state model inference	258
9.4	Distributed State Model inference	259
9.5	Online oracles	261
9.6	Delta GUI change detection oracle	262
9.7	Academia-industry collaboration	265
9.8	Scriptless test automation is a complementary solution	266
9.9	Artificial Intelligence	267
10	Conclusions and future work	269
10.1	Contributions	269

10.2 Publications 272

10.3 Future work 277

References **281**

List of figures

1.1	A tester who exercises a software system while gathering evidence to make informed decisions about its quality characteristics	5
1.2	Overview of an open financial services architecture	6
1.3	Unit tests testing deposit and withdrawal methods	7
1.4	Integration testing of Banking and Remote Communication components . .	8
1.5	System testing interaction that invokes underlying methods and interfaces .	9
1.6	Scripted testing approach example with 3 test scripts	13
1.7	Scriptless testing flow overview	15
1.8	TESTAR modular architecture overview	17
1.9	TESTAR scriptless thesis objectives	19
2.1	The widget tree of a Desktop application that constitutes the GUI state . . .	28
2.2	The widget tree of a Web application that constitutes the GUI state	28
2.3	The widget tree of a Mobile (Android) application that constitutes the GUI state	29
2.4	Taggable Widget, State, and Action classes obtained from a System Under Test	30
2.5	Diagram of action derivation logic for interactive widgets	33
2.6	LabRecruits game scenario	40
2.7	LabRecruits-plugin overview	41
2.8	The LabRecruits entities that constitute the game state	42
2.9	Representation of the observed LabRecruits environment by the agent . . .	42
2.10	Taggable IV4XRWidgetEntity, IV4XRState, and IV4XRAction classes obtained from a game-SUT plugin that implements the IV4XR framework interfaces	43
2.11	LabRecruits available <i>navigation</i> mesh positions	45

3.1	Dynamic widgets properties of a desktop GUI and XR game systems that can constantly change without performing user or automated interaction . . .	49
3.2	Dynamic widgets properties of a desktop GUI and XR game systems that can constantly change after each automated or user interaction	50
3.3	A concrete action identifier based on typed text character tends to infinity depending on the maximum number of characters allowed	50
3.4	Example of web widgets properties from Parabank that can be selected for the main abstraction mechanism	52
3.5	OBS non-determinism when deciding to ignore the dynamic Widget Title property	53
3.6	Example of inferred state model from Parabank web application	54
3.7	Example of surjective abstract and concrete states mapping from Parabank web application	56
3.8	Unvisited actions first ASM using the state model information	57
3.9	Rachota Timetracker SUT	60
3.10	The code coverage (%) that was reached when comparing the Random_ASM with 4 different abstraction levels of the State_Model_ASM	61
3.11	The number of abstract states (top layer) and concrete states (mid layer) . . .	62
3.12	The decision challenge of excluding or including the widget Title property in Rachota	64
3.13	Web widget in Parabank that uses a dynamic Web Id property	64
3.14	Desktop widget (table) in Rachota that contains dynamic widgets (table rows)	65
3.15	Web widget (table) in Parabank that contains dynamic widgets (table rows)	65
3.16	Parabank non-deterministic transition after executing the same action in the same particular state	66
3.17	Abstraction strategy used by TESTAR which combines a main abstraction mechanism with abstraction sub-strategies	67
3.18	Implemented map that associates grammar action expressions with TESTAR actions	70
3.19	An example of a WebformSUT form	76
3.20	Architecture of WebformSUT experiment	77
3.21	Parabank customer care and find transactions forms	78
3.22	Architecture of Parabank experiment	80
3.23	WebformsSUT - success rates	81
3.24	WebformsSUT - actions per field deviation	82
3.25	Customer care form - success and unsuccess submits	83

3.26	Customer care form - random form-filling deviation	84
3.27	Customer care form - human-like form-filling deviation	84
3.28	TESTAR operational flow with LabRecruits	87
3.29	LabRecruits spatial coverage heat map	91
3.30	Observed and interacted buttons coverage for LabRecruits	92
3.31	Observed and walked floor positions coverage for LabRecruits	93
4.1	Distributed state model inference architectures	99
4.2	Shared state model knowledge ASM used by multiple distributed TESTAR instances	102
4.3	Dealing with non-deterministic models	105
4.4	Architecture for the execution of distributed experiments	114
4.5	Parabank code coverage comparison for 1 and 6 groups of Dockers. Each line represents one of the thirty repeated executions.	115
4.6	Parabank state model comparison for 1 and 6 groups of Dockers. Each line represents one of the thirty repeated executions.	116
4.7	Parabank average code coverage growth for each group of Dockers.	116
4.8	Shopizer code coverage comparison for 1 and 6 groups of Dockers. Each line represents one of the thirty repeated executions.	119
4.9	Shopizer state model comparison for 1 and 6 groups of Dockers. Each line represents one of the thirty repeated executions.	120
4.10	Shopizer average code coverage growth for each group of Dockers.	120
5.1	Error message displayed in the GUI of the UPV	128
5.2	Error message displayed in the buffer of a SUT process	129
5.3	JavaScript error messages displayed in the browser console of the UPV	129
5.4	Unmatching values GUI failure in an Amazon web page	131
5.5	Repeated text GUI failure in an Amazon web page	132
5.6	OBS Issue: "Split Recording File" hotkey increases when changing profiles	132
5.7	UPV misspelling issues related to an accent character and English translations	133
5.8	OBS Issue: Dialog box text overwrites the Info badge	135
5.9	UPV feature that sanitizes special characters but is vulnerable to HTML encoding	138
5.10	Test oracles that check Space Engineers game features	140
5.11	Output Structure for Test Results	143
5.12	State Model analysis interactive graph	143
6.1	Overview of a delta testing process in software projects	146

6.2	Search query to embrace diverse terminology for GUI delta changes	147
6.3	Overview of the process for the systematic mapping literature review	149
6.4	Representation of a GUI state model of a desktop application that contains 7 states and 11 action transitions	159
6.5	Partially complete model inferred obtained of a desktop application due to a limited set of actions with a restricted depth	160
6.6	OBS partial inferred state model from SM_{new} (v30.0.2) and SM_{old} (v29.1.3)	161
6.7	Merge graph technique for visual graph comparison with OBS partial inferred state models	164
6.8	Visual representation of changed states	165
6.9	Merge graph technique comparison using state and action abstract identifier vs. action description	166
6.10	Architecture of GUI Delta Change Detection experiments	174
6.11	Interactive web interface of the GUI change tool that creates a merged partially complete model from the OBS SM_{new} (v30.0.2) and SM_{old} (v29.1.3) versions with a depth of 3 actions	177
6.12	Static widget changes challenge: The new Open Scene Filters icon-button added in the Scenes dock panel from v28.1.2 to v29.1.3 introduces <i>noise</i> in all states since they are all marked as changed	181
7.1	The architecture of the continuous TESTAR piloting environment	193
7.2	Prodevelop CI/CD Pipeline	196
7.3	Integration of TESTAR through an API in a distributed environment	200
7.4	Posidonia and TESTAR Logs Structure	201
7.5	TESTAR test cycle extended with forced actions	204
7.6	TESTAR integration with the ING mobile application	212
7.7	Figure showing the top 20% of the packages with the most difference in coverage between TESTAR and Espresso	215
7.8	Space Engineers game scenario	219
7.9	Space Engineers-plugin overview	221
7.10	Space Engineers observed environment by the agent	222
7.11	Navigable actions in Space Engineers to reach interactive entities	223
7.12	TESTAR operational flow with Space Engineers	224
7.13	Space Engineers spatial coverage map	228
7.14	Observed and interacted blocks coverage	230
7.15	Observed and walked floor positions coverage	231

8.1	Space Engineers development cycle diagram	239
8.2	Overview of the Space Engineers-plugin and Kotlin test script example . . .	241
8.3	Space Engineers Kotlin test script with GWT structure	244
8.4	Cucumber mapping of Space Engineers-plugin functions with GWT statements	245
8.5	Space Engineers BDD test scenario to validate astronaut inventory components	246
8.6	Space Engineers pre-designed game level for testing	246

List of tables

3.1	Java Access Bridge properties and the possible impact of using the attribute for state abstraction in Rachota	59
3.2	Details of Rachota SUT	60
3.3	Number of p-values < 0.05 for WebformsSUT.	80
3.4	Mann-Whitney U p-value significant difference	82
3.5	Wilcoxon p-value significant difference for LabRecruits	94
4.1	Preliminary results of Shared_Knowledge_ASM	102
4.2	Details of the selected SUTs	108
4.3	Independent time variables	109
4.4	Independent variables for smarter action derivation	110
4.5	Independent variables for the main abstraction mechanism	111
4.6	Independent variables for abstraction sub-strategies	112
4.7	Effort time to design the independent variables	113
4.8	Parabank Kruskal Wallis results	117
4.9	Parabank significant difference at 360 seconds Wilcoxon: p-value (p), Cliff's δ : small(S), medium(M) and large (L) difference	117
4.10	Parabank significant difference at 480 seconds Wilcoxon: p-value (p), Cliff's δ : small(S), medium(M) and large (L) difference	117
4.11	Parabank significant difference at 600 seconds Wilcoxon: p-value (p), Cliff's δ : small(S), medium(M) and large (L) difference	118
4.12	Parabank Consumption Average	118
4.13	Shopizer Kruskal Wallis results	119
4.14	Shopizer significant difference at 500 seconds Wilcoxon: p-value (p), Cliff's δ : small(S), medium(M) and large (L) difference	121
4.15	Shopizer significant difference at 1750 seconds Wilcoxon: p-value (p), Cliff's δ : small(S), medium(M) and large (L) difference	121

4.16	Shopizer significant difference at 3000 seconds Wilcoxon: p-value (p), Cliff's δ : negligent(N), small(S), medium(M) and large(L) difference	122
4.17	Shopizer Consumption Average	122
6.1	GUI Change Detection research papers obtained from the systematic mapping of the literature	152
6.2	Details of the selected SUT Objects	168
6.3	Independent variables for sequences inference strategy	169
6.4	Independent variables for action derivation strategy	170
6.5	Independent variables for the main abstraction mechanism	171
6.6	Independent variables for abstraction sub-strategies	172
6.7	Effort time to design the independent variables and run pre-executions	173
6.8	State Model Inference results	176
6.9	OBS-Studio change detection qualitative results	177
6.10	Calibre-Web change detection qualitative results	181
6.11	MyExpenses change detection qualitative results	185
7.1	Summary of existing tools for scriptless mobile testing	209
7.2	Coverage results for the Android application	215
7.3	Wilcoxon p-value significant difference for Space Engineers	230
7.4	Implemented TESTAR test oracles that have detected new bugs in Digioffice	235

Abbreviations and Acronyms

AI: Artificial Intelligence
ASM: Action Selection Mechanism
ASR: Action Selection Rules
BDD: Behavior-Driven Development
BFS: Breadth-first search
CDN: Change Detection and Notification
CI: Continuous Integration
CLI: Command Line Interfaces
CNN: Convolutional Neural Network
DMIS: Distributed Model on Independent SUT
DMSS: Distributed Model on a Shared SUT
DNN: Deep Neural Networks
DOM: Document Object Model
DSL: Domain Specific Language
EBNF: Extended Backus-Naur Form
GP: Genetic Programming
GUI: Graphical User Interface
GWT: Given-When-Then
IoT: Internet of Things
LOC: Lines Of Code
ML: Machine Learning
NavMesh: Navigation Mesh
NL: Natural Language
OCR: Optical Character Recognition
OS: Operating System
QA: Quality Assurance
RL: Reinforcement Learning
SDLC: Software Development Life Cycle

SUT: System Under Test

VM: Virtual Machine

XSS: Cross-Site Scripting

XR: eXtended Reality

Chapter 1

Introduction

Software has become a central pillar in the daily lives of people worldwide. The digital economy and e-commerce [295, 102, 86], Industry 4.0 processes [149], healthcare management and clinical support systems [303, 192], video streaming services [210], games for education or entertainment [64], and numerous other fundamental daily activities are directly or indirectly influenced by a multitude of interconnected software technologies and services.

Before the 2000s, the growth, use, and vision of future software dependency, now a reality, were encapsulated in the well-known phrase: *Software is everywhere* [31, 57]. Researchers, professors, artists, analysts, and experts from various fields highlighted how industry and society's daily life were adopting software technology for telecommunications, transportation, government management, music, literature, healthcare, agriculture, national defense, banking, and virtually any human need, whether for work or leisure.

By 2011, the widespread adoption of software by industrial companies and users was evident on both economic and social levels. Tech giants like Google, Facebook, Apple, LinkedIn, Amazon, Microsoft, and others were expanding globally, enhancing their services, reducing production costs, and enabling users to remotely access Internet-based services for searches, knowledge sharing, office productivity, books, management, recruitment, music, movies, video games, and more. This growth and adoption of software, as well as its optimistic future forecasts, led to the assertion that: *Software is eating the world* [15].

Today, society and new generations no longer consider how software will be used or how it will disrupt and change their environment; software use has become a habit in human society. Instead, modern concepts like Industry 5.0 [151] and Society 5.0 [75] intrinsically accept the use of software and focus on envisioning how future societies will adopt new technologies such as the Internet of Things (IoT), Artificial Intelligence (AI), big data, and robotics to enhance industrial production processes and address social issues by integrating cyberspace and physical space.

Software is present in modern everyday life. It is there when the alarm rings in the morning to wake you up and when you connect to the Internet through your router to check messages, news, or reels on your mobile phone. We rely on software; we want it to work, and we do not want it to fail. A malfunctioning alarm could cause you to miss important events or appointments. Even worse, a failure in the router software could lead to one of the most frustrating experiences of the 21st century: disrupt your internet connection [107].

Consequently, software Quality Assurance (QA) is paramount for society to trust the software they employ in their daily lives.

1.1 Software Quality

In a formal definition, *Software quality is achieved by conformance to all requirements regardless of what characteristic is specified or how requirements are grouped or named* [42]. From a user's perspective, software quality is determined by the software's capability to do what it is supposed to do while performing the tasks correctly [272]. When the software meets these expectations, user satisfaction increases. Then, the greater the level of user satisfaction, the more likely the user is to say the software is of high quality [76].

Software is created to accomplish one or multiple objectives. For example, a mobile banking application is designed to allow users to consult their savings and manage transactions. Software requirements must be correctly implemented in software functions to achieve these objectives. Moreover, it is crucial to perform these functions without issues. For instance, when sending a transaction, the application must securely decrease the sender's balance and increase the receiver's balance. Throughout this transaction process, the application must not crash, hang, or slow down, and the updated information should be accurately displayed in the savings activity. If this mobile banking application allows users to perform all these tasks without finding problems, they will be satisfied and consider the software high quality.

Ideally, organizations strive to develop high-quality software for their users. Nevertheless, any task involving humans, including software development, carries the inherent risk of suffering mistakes. These mistakes can lead to faults in the software, negatively impact user satisfaction, and make users consider the software is of poor quality. For example, a mobile banking application may provoke a negative user satisfaction if it takes a long time to load, it is not easy to use to make a transaction, or at some point in time, a data breach informs users their personal data has been leaked.

Software Failures

The phrase *Software fails* is a concise and perhaps more realistic reflection of the current state of today's software systems than *Software can fail* [83, 57, 78, 256]. This does not imply that all software systems experience critical failures or become unusable, but it results in a decrease in the quality of software.

Numerous research studies [109, 205, 78] and real-world examples illustrate poor software quality due to failures. Let's consider the following recent examples of news stories involving banking and game applications.

As examples of internal factors, in 2021, Santander Consumer Bank faced a misconfiguration failure when private files were accidentally indexed by search engines¹. In 2023, a malfunction in the Zengin Data Telecommunication System resulted in a two-day system outage affecting 5 million bank transactions in Japan². In another case in 2023, the Bank of Ireland experienced a "technical issue" glitch that allowed customers to withdraw or transfer money beyond their monetary limits³.

Regarding game environments, No Man's Sky and Fallout 76 were two titles released in 2016 and 2018, respectively, that suffered from bugs and other issues, leading to significant discontent among their player bases⁴. Similarly, Cyberpunk 2077 was officially launched at the end of 2020 and was plagued by disruptive functional, visual, and performance bugs, prompting Sony to offer full refunds to those affected users⁵.

External factors can also contribute to software failures. In 2023, a third-party software vulnerability led to Deutsche Bank and Postbank experiencing an attack where criminals accessed customers' private bank data⁶. Moreover, in 2022, Andorra country suffered a denial of service attack by a group of people with the intention of disturbing a Minecraft tournament⁷. This attack knocked down the country's internet, indirectly affecting other banking software connections.

Except for the Bank of Ireland glitch, which users might consider a desired feature rather than a failure, neither software owners nor customers want to suffer the consequences of software failures. Software failures range from minimal repercussions like user frustration to significant economic and social impacts.

¹<https://cybernews.com/security/one-of-biggest-european-banks-leaking-sensitive-data-on-website/>

²<https://japannews.yomiuri.co.jp/business/companies/20231012-142639/>

³<https://www.nytimes.com/2023/08/16/world/europe/bank-ireland-technical-glitch.html>

⁴<https://www.forbes.com/sites/erikkain/2018/10/24/it-sure-seems-like-fallout-76-will-be-the-no-mans-sky-of-2018/>

⁵<https://www.nytimes.com/2020/12/18/technology/cyberpunk-2077-refund.html>

⁶<https://cybernews.com/security/deutsche-ing-postbank-impacted-moveit-hack-clop/>

⁷<https://www.wired.com/story/andorra-ddos-minecraft-nso-group-security-news/>

As described in the examples above, failures in financial software can have substantial economic impacts, resulting in million-dollar losses for financial institutions and instability for the monetary savings of their customers [78]. This instability can lead to severe social repercussions for the affected customers. However, the most critical impact of a software failure can risk human life.

Aviation, with nearly 100,000 commercial flights daily, is crucial for economic and social activities. Despite the high standards for aviation engineering and software prioritizing public safety, some aviation companies are more concerned about cost and schedule than passenger safety. Unfortunately, software engineering failures in the Boeing 737 MAX aviation system, due to a failure of an Angle of Attack sensor and the subsequent activation of new flight control software, led to the crash of two airplanes and the deaths of 346 passengers [116].

Healthcare systems are another critical area where software plays a vital role in measuring, monitoring, and managing patients' health. For more than 30 years, software has become essential in complex healthcare systems, and for more than 20 years, it has been reported that software failures can cause malfunctions in medical devices [284]. While software aids in detecting diseases, designing treatments, and managing patients, failures in these healthcare systems have resulted in the loss of human life [139].

A study analyzing Health Information Technology reports submitted to the US Food and Drug Administration Manufacturer and User Facility Device Experience database from January 2008 to July 2010 revealed significant software-related issues [162]. Of the 436 incidents examined, 96% were machine-related issues, with more than 40% being software problems. One notable issue involved a user interface that didn't provide medication doses in milligrams, leading to the administration of three times the maximum dose of a drug, resulting in acute renal failure and death. Additionally, the poor functionality of the software caused a missed opportunity to diagnose and treat life-threatening diseases, contributing to the patient's death.

Summarizing, software has become an indispensable component of daily life in modern societies. Given this dependency, any decrease in software quality and the occurrence of software failures can lead to significant economic and social impacts⁸. Therefore, it is crucial to dedicate substantial efforts to ensure software quality in all software systems, as software now impacts nearly every aspect of our lives. Although quality is an inherently ambiguous concept and often difficult to define, software quality encompasses characteristics such as functionality, performance, compatibility, security, reliability, usability, portability, and maintainability [89]. These characteristics, however, are not absolute and are not always

⁸<https://www.techtarget.com/whatis/feature/Explaining-the-largest-IT-outage-in-history-and-whats-next>

easy to quantify or measure. Their relevance and the criteria used to assess them can vary depending on the software's objectives and the intended use of its stakeholders [142, 99].

1.2 Software testing

Software testing is the most widely used process to assure software quality [193, 120]. The purpose of testing is to evaluate software quality characteristics by exercising a software system while collecting evidence to make informed decisions (see Figure 1.1) [281].

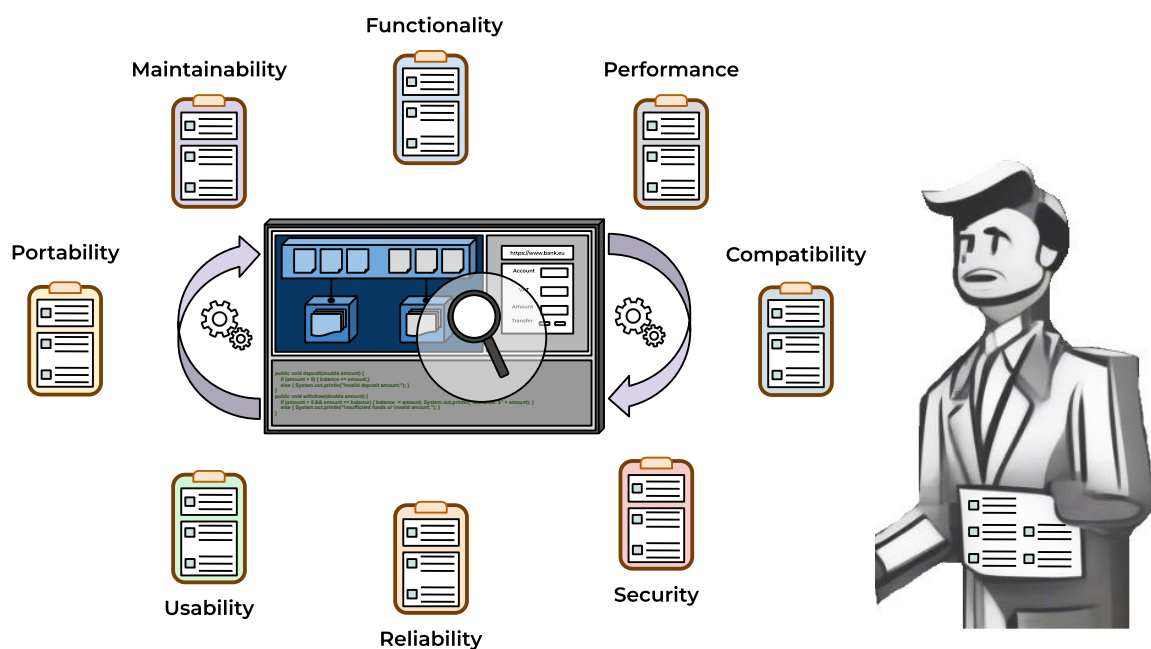


Fig. 1.1 A tester who exercises a software system while gathering evidence to make informed decisions about its quality characteristics ⁹

Testing is an essential part of the Software Development Life Cycle (SDLC). The SDLC describes the procedure for developing software [281] and is typically represented through various SDLC models, such as the Waterfall model, the V-model, the Iterative mode, the Spiral model, and the Agile model [25, 134, 255, 106]. Each model has its own advantages and disadvantages, requiring companies or project managers to evaluate and choose the most suitable one for their software projects. Regardless of the chosen SDLC model, testing during software development is primarily, though not exclusively, conducted at three levels: unit testing, integration testing, and system testing [53, 42, 121]. Figure 1.2 shows an overview

⁹Tester image generated using SD-XL 1.0-base Model Card.

of an open financial services architecture based on the use of intelligent mobile devices on which the three testing levels can be differentiated [147].

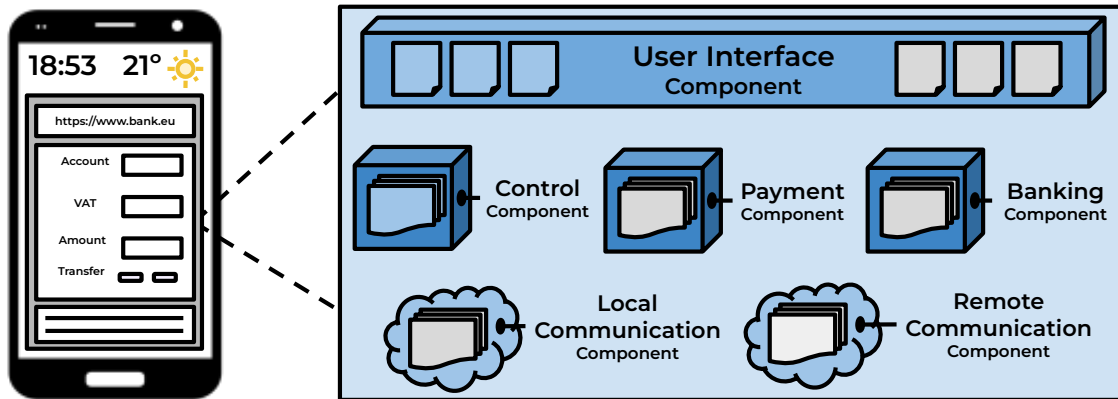


Fig. 1.2 Overview of an open financial services architecture

Unit testing

Unit testing, also known as component testing, is the testing implemented at the code level. Software applications consist of components composed of functions and methods that initialize or modify variables and objects to accomplish specific functions or procedures. These methods might have diverse modular objectives, such as the User Interface Component, which is responsible for presenting information to the user; the Payment Component, which includes algorithms for account or credit card-based payments as well as algorithms for digital cash transactions; the Banking Component, which includes services for accounting and personal information, bank transfers, investments, and application forms; or the Remote Communication Component, which controls the mobile device's communication when a remote transaction, such as the provision of a retail banking service and or an Internet payment, is taking place.

Unit tests aim to ensure code quality by checking the correct initialization, modification, increase, or decrease of the values of variables and objects within the methods of the classes that make up these components. This is typically achieved by providing specific inputs to these methods and then using assertions to compare their actual outputs with the expected output results. For example, consider a desired banking feature to manage deposits and withdrawals. Then, as shown in Figure 1.3, unit tests can check that this listing transaction method in the Banking Component: (1) correctly update the balance when a valid deposit is made, ensuring that the balance increases by the deposited amount if the deposit is greater than zero; (2) prevent withdrawals that exceed the available balance, returning an error

message and leaving the balance unchanged; (3) reject invalid deposit amounts, such as negative values, and display an error message without changing the balance.

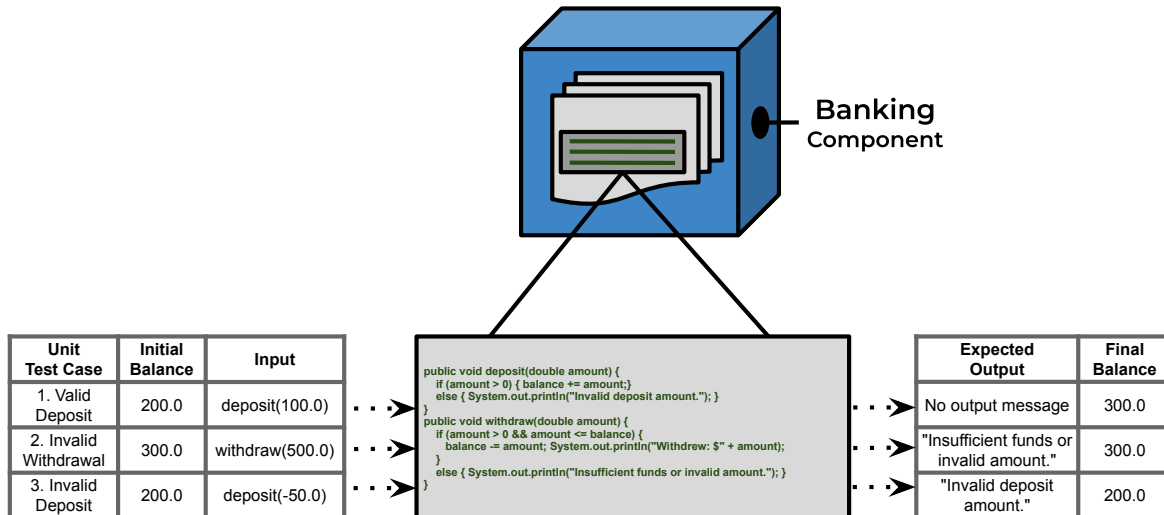


Fig. 1.3 Unit tests testing deposit and withdrawal methods

Writing unit tests also help in understanding the structure, readability, and maintainability of software code components. This practice can guide refactoring and isolating methods, classes, or even new components, thereby improving code quality. For example, instead of implementing in the same class a set of methods to perform remote communication with the central bank database to query customer transactions and display the information in the user interface, isolating the user interface from the banking listing feature and the network database communication makes it easier to identify failures and facilitates future software maintenance and updates.

Integration testing

While unit testing targets individual methods and classes, integration testing examines the communication between the methods and classes across different components. When two or more components, such as the User Interface Component, the Banking Component, and the Remote Communication Component, are designed to interact, integration testing looks for defects in the use of the component interfaces and in the data sent during communication between components.

For example, as shown in Figure 1.4, integration testing checks if the user's transaction history retrieved by the Banking Component via the Remote Communication Component is accurately displayed by the User Interface Component. This involves testing that the Banking Component correctly sends a request to the Remote Communication Component. The data

must be properly formatted and transmitted back to the Banking Component, which then passes it to the User Interface Component for display. This type of testing identifies issues that arise from the interaction of components, such as data mismatches or communication errors, which may not be evident in unit testing.

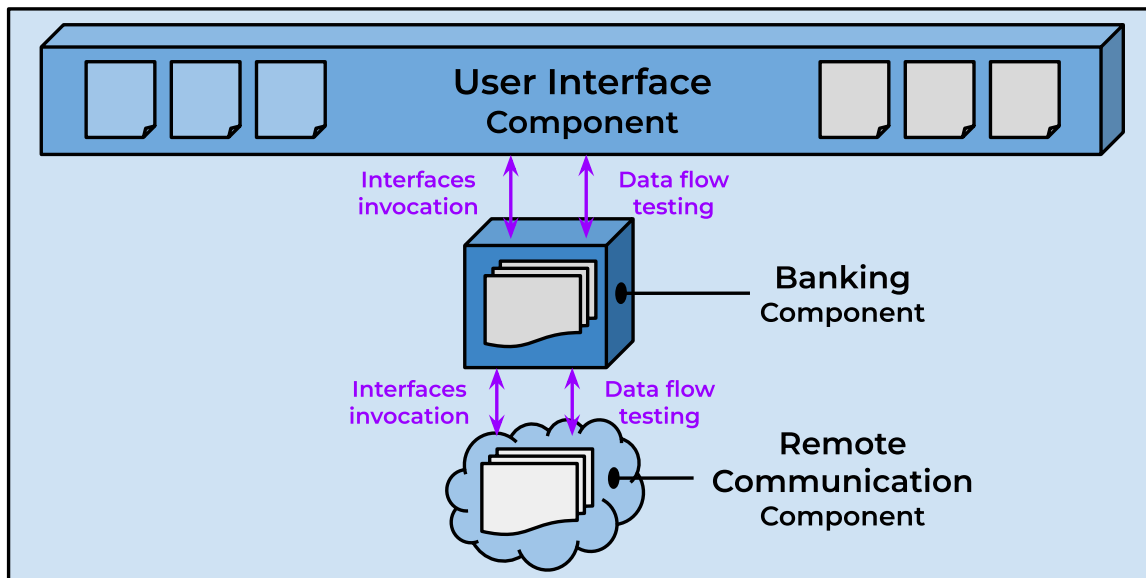


Fig. 1.4 Integration testing of Banking and Remote Communication components

System testing

System testing examines the quality of the entire software system when all components are integrated and deployed in a final software product. This testing involves ensuring the quality of the functional aspects of the system from the user's perspective and other non-functional aspects like the security of the software or whether it is easy for users to understand.

For example, as shown in Figure 1.5, system testing simulates real-world user interactions with the graphical user interface (GUI), such as logging into the bank application and accessing the banking transfer window. If the GUI correctly displays transaction information, if the login and access times are acceptable to users, and if the GUI is not easy to use for performing transactions, users will likely be satisfied and perceive the software as high quality.

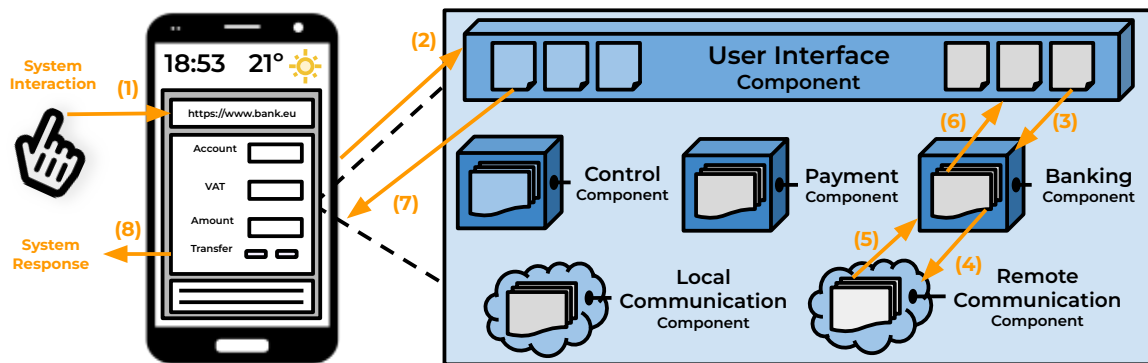


Fig. 1.5 System testing interaction that invokes underlying methods and interfaces

1.3 Graphical User Interface testing

The Graphical User Interface (GUI) represents the main interaction point between software and its end users, making GUI testing a fundamental method for testing desktop, web, and mobile applications at the system level [26, 241]. Since graphical user interfaces are the primary means through which end users perceive and interact with software systems, and because testing through the GUI helps evaluate the quality of the entire software product, system GUI-based testing plays an essential role in the software industry [157, 185].

Test Oracle

An oracle in software testing is a mechanism (which could be a human or a machine agent) that determines whether software behaves as expected under specific conditions during a test. Depending on whether the output test results are according to the expected output, the oracle provides a "pass" or "fail" verdict [27, 28, 42].

Test oracles are implemented under varying conditions depending on the specific software being tested, the level of testing, and the type of testing approach. For example, at the system GUI level, a security test oracle might check that the GUI displays a "session expired" message when attempting to retrieve customer information after the user's login session has timed out. An accessibility test oracle could confirm that all images displayed in the GUI contain alternative text describing the image content. An invariant-based test oracle might ensure there is always a static and visible top bar in the GUI for users to navigate between functions in an application. Additionally, a functional test oracle might aim to check that the banking transfer window lists some transactions after the users log in to the system.

Test Adequacy criteria

Despite the existence of diverse approaches and techniques aimed at detecting software failures and enhancing software quality, it is not possible to ascertain the presence, quantity, and nature of faults in software. When a failure is detected, it indicates the existence of software faults, but it does not reveal all present faults [77]. Conversely, if no failure is detected, it cannot be assumed that the software is fault-free.

Given the size and complexity of software, it becomes necessary to decide when to cease testing, even if the absence of faults cannot be entirely ascertained. Furthermore, software may be used by different people to accomplish different objectives, making testing dependent on context and intended use.

The test adequacy criteria serve to determine whether sufficient testing has been performed and if testing can be concluded [309]. If all planned tests have been executed, the test adequacy criteria may help indicate whether the testing process is complete. However, even if the testing process is deemed complete, it is not possible to have a real measure of the quality of these tests.

For this reason, it is necessary to use a *surrogate measure* that helps to estimate the quality of the testing process. This will not directly measure software correctness or the absence of bugs but will provide an approximation of the extent to which the software has been exercised (i.e., covered) during testing [281].

Test coverage criteria indicate which parts of the software have been covered or remain uncovered. With the results from these criteria, it is possible to indicate if the quality of the testing process is sufficient because it covers the desired software features to test or if more testing is required. Some of the most commonly used coverage criteria in software GUI testing include:

- **Code Coverage:** This criterion measures the extent to which the source code of a program is executed during testing [115].
- **GUI Coverage:** This criterion evaluates which GUI elements were identified and interacted with during testing [177].

1.3.1 Manual test execution

In software testing, human testers design and execute test cases by interacting with the System Under Test (SUT) while observing the software results and acting as a human oracle [254]. A test case typically includes a set of conditions or variables that indicate how to interact with the software and which test oracles (e.g., functional, security, performance,

accessibility, etc., test oracle) to apply to determine if tests pass or fail. Testers typically perform documented and exploratory testing at the system level by simulating end-user GUI interactions. Documented test cases can serve as formal checklists for test execution or as high-level descriptions [123]. Conversely, exploratory testing relies on testers' experience and tacit knowledge to design test cases that identify potential weak areas or unexpected interactions not covered by documented tests [124].

Human testers are indispensable in software testing. Human testers rely on tacit knowledge to design test cases, which are essential for assuring software quality. This tacit knowledge includes comprehension, intuition, and accumulated experience [283], which is crucial for determining whether the software meets functional and non-functional requirements and satisfies the customer's needs. Numerous studies indicate that the efforts of professional testers and their domain-specific tacit knowledge significantly impact test case design, testing results, and the overall quality of the software [123, 124].

However, while human testers are indispensable and highly effective due to their irreplaceable tacit knowledge, the testing effort can become particularly tedious when it is necessary to continuously repeat the same test case executions. Nowadays, the dynamic and rapid evolution of SDLC practices makes software systems to be continuously modified [209, 117, 97]. This requires testers to perform regression testing and re-executing documented test cases, repeatedly performing the same interactions and inspecting software outcomes to make sure that recent code changes have not adversely affected existing functionality [281]. Consequently, relying exclusively on manual efforts in such regression testing scenarios can lead to heavy workloads, time constraints, frustration, and fatigue. This, in turn, can negatively impact the overall testing process and software quality, as manual testing time constraints may lead to a lack of attention during the testing process.

To address these challenges, test automation has been researched and developed to complement and support human testers.

Test automation involves creating a mechanically executable representation of a test case [271]. This can be achieved by formulating in test cases which interactions to execute and which test oracles to use to check software outcomes. Automated test cases are typically interpreted and driven by specialized software programs. In essence, test automation involves the use of special software tools to control the execution of tests and then compares actual test results with predicted or expected results [195].

Automation can streamline testing processes, reduce workloads, and provide valuable information to enhance tester productivity. By combining human tacit knowledge with automated testing, organizations can create a more effective and efficient testing approach, achieving higher software quality.

1.3.2 Scripted testing approach

The scripted testing approach aims to automate regression test cases to reduce the testers' effort involved in repetitive testing tasks. Test scripts automate the execution of test case interactions and oracles, which determine whether a test script has passed or failed. These scripts can be manually written by coding the test case conditions or using visual images [45, 12], recorded with capture and replay techniques while a human manually performs the test case interactions [131, 152], or generated from manually created models that represent a set of test cases for specific software [136, 51].

Figure 1.6 illustrates the execution of a set of test scripts. In this specific test scenario, a subset of three scripts are executed under specific conditions to test specific bank software system features.

The first script connects to the application, clicks the button to access banking services, clicks the tab to list user expenses transactions, and applies a test oracle to assert that the application shows two transactions in the user interface, which are formatted in a specific way. The second script connects to the application, clicks the button to access the payment service, fills in the account and amount fields but leaves the VAT field empty to simulate an incomplete form, clicks the button to send the payment, and applies a test oracle to assert one pop-up message as appeared indicating the user to fill the VAT field. The third script connects to the application, clicks the button to access the payment service, fills all fields with valid text values to complete a form, clicks the button to send the payment, and applies a test oracle to assert that the application indicates a successful payment and displays a confirmation message.

The automated scripted approach for regression testing offers several benefits, such as reducing human effort and costs and playing an essential role in maintaining the software quality by assuring that software changes do not introduce unintended failures that break existing functionality [230]. However, like many test automation approaches, it also presents several limitations that require further research [294, 195]. For instance, to reduce maintenance costs associated with constantly updating and fixing broken scripts, ongoing research explores the use of various software element locators and visual technologies to improve the robustness and reliability of test scripts [306, 94, 110, 262, 264, 196, 92].

In this thesis, we acknowledge the significance of test automation for reducing repetitive, boring, and error-prone regression testing efforts through scripted solutions. Nevertheless, we emphasize that human and automation efforts should not focus solely on creating and maintaining a vast number of test scripts. Given the nearly infinite interaction paths and testing scenarios within large, complex software systems, a scripted regression testing

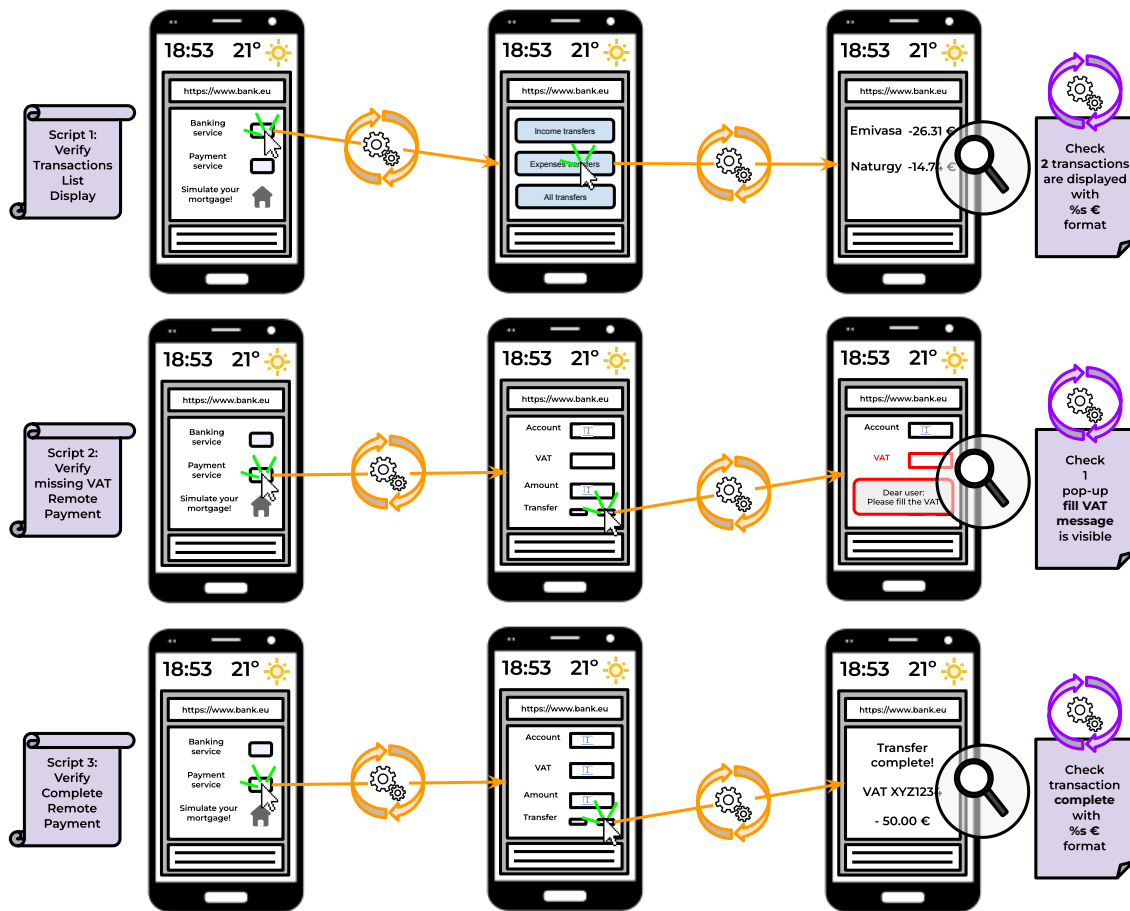


Fig. 1.6 Scripted testing approach example with 3 test scripts

strategy should prioritize verifying critical functional paths rather than attempting to achieve exhaustive coverage through excessive test scripts.

Considering the size and complexity of modern software applications, achieving high test coverage solely through scripted testing would require an enormous and impractical effort. Therefore, it is necessary to explore and integrate other automated testing approaches, such as scriptless testing for exploratory testing. These scriptless testing techniques have the potential to address uncovered areas and uncover unexpected failures by exploring unforeseen states and performing unanticipated combinations of interactions.

1.3.3 Scriptless testing approach

Scriptless testing is an approach that employs explorative algorithms to dynamically generate and automatically execute test sequences. Unlike the scripted approach for regression testing, scriptless testing does not rely on scripts with predefined sequences of actions. Instead, it

aims to explore the software by making runtime decisions as available states and actions are discovered.

The most basic form of scriptless testing is also known as monkey testing, which involves randomly exercising a software program using an automated test tool [119]. There are two main types of monkey testing: dumb monkey and smart monkey [202]. A *dumb monkey* has no knowledge of the software system and performs purely random actions while checking that the software is robust and does not crash or hang. In contrast, a *smart monkey* has a basic understanding and knowledge of the software and testing objectives and employs action selection algorithms with memory and a degree of intelligence to make smarter exploratory decisions. For example, in scriptless GUI testing, smart monkeys can employ GUI ripping [176] and state model inference techniques [167] to identify and maintain memory of discovered GUI states and actions or use Machine Learning (ML) algorithms to learn how to test more efficiently [163].

Scriptless testing execution flow

The scriptless testing approach, regardless of the degree of intelligence involved, always introduces a degree of randomness and variability when exploring a software system. This randomness can uncover states not covered by scripted tests and identify unexpected software failures resulting from unanticipated combinations of interactions. Additionally, because scriptless testing does not require predefined test scripts to automatically interact with and test software requirements, it can potentially cover parts of regression test cases and help reduce the effort needed for test script maintenance.

Figure 1.7 provides an overview of the scriptless testing approach. Unlike the scripted approach, the exploratory scriptless execution follows a cyclical flow instead of a sequential execution of actions.

- First, and depending on the type of System Under Test (SUT), scriptless tools rely on various technologies to connect with the SUT, obtain information about the elements that define the state of the application, and determine the set of possible action interactions that can be executed in that state.
- Second, scriptless tools employ an action selection algorithm, ranging from purely random selection to more sophisticated intelligent algorithms, to select and execute one action. Moreover, these action selection algorithms may be state model-based, which maps how to navigate to discovered states by executing action transitions.

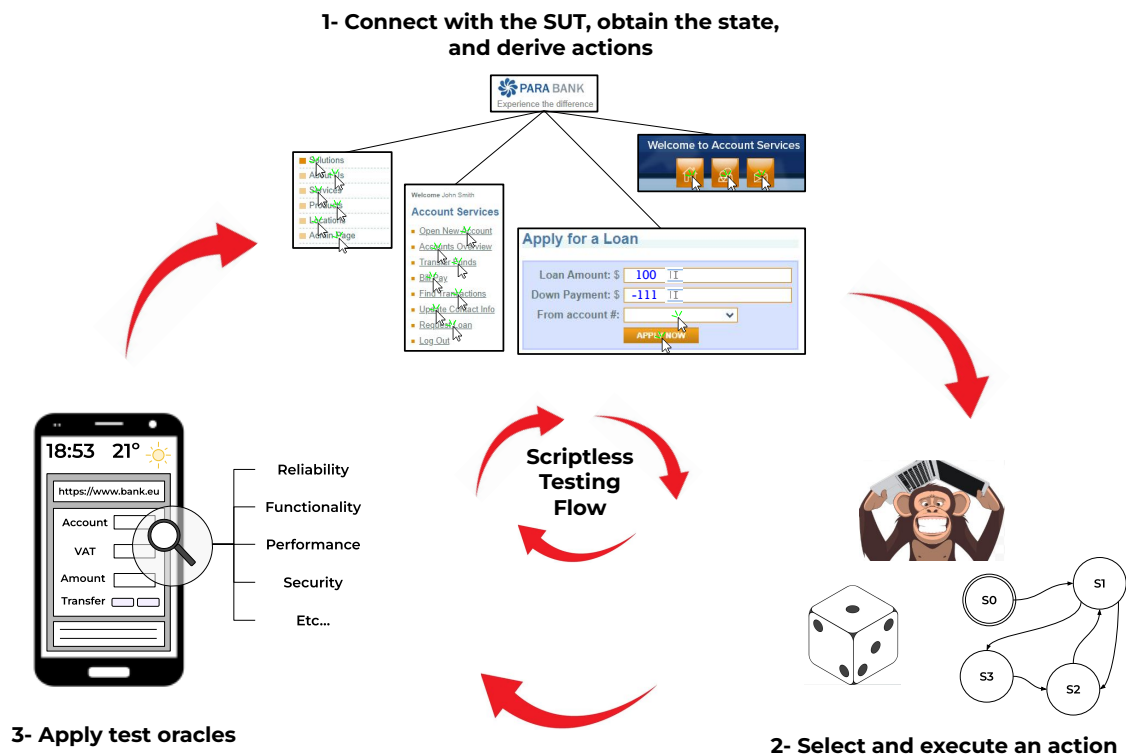


Fig. 1.7 Scriptless testing flow overview

- Third, scriptless tools apply test oracles to determine if the discovered states of the application contain any failures. These test oracles may vary depending on the SUT and the type of testing approach.

Since exploratory scriptless testing does not have a predefined number of actions to execute, this cyclical execution flow typically continues until the tester decides the tool must stop. This decision can be based on reaching a maximum number of executed actions, a maximum execution time, or a specific number of discovered failures.

1.4 Scriptless testing tools

There exist several test automation tools that rely on various GUI technologies, including accessibility APIs, automation frameworks, and image recognition techniques, to:

1. Connect with the SUT, obtain information about the elements that define the GUI state of the application, and determine the set of possible action interactions that can be executed in that state;

2. Employ an action selection algorithm ranging from purely random selection to more sophisticated intelligent algorithms, to select and execute one action;
3. Apply different types of test oracles to determine if a failure has been discovered.

For desktop systems, Murphy [8] uses UIAutomation and image recognition, GUI Driver [6] uses the Jemmy Java library, and both AutoBlackTest [169] and AUGUSTO [170] use IBM Functional Tester. For web systems, Selenium WebDriver is the commonly used technology, as seen in ATUSA [180], Crawljax [181], and Webmate [72], though other frameworks such as Cypress, used in Cytession [186], are also gaining popularity. For mobile systems, scriptless tools such as Stoa [265], DroidBot [155], CrawlDroid [52], DroidMate-2 [41], and AutoDroid [2] primarily focus on testing SUTs in Android systems using technologies such as Android Debug Bridge (ADB), Android UI Automator, or Appium. Finally, some scriptless tools like GUITAR [199] integrate various GUI technologies to enable testing in multiple desktop, web, and Android environments.

Despite the existence of these scriptless testing tools, they face various limitations:

1. They are often restricted to specific types of SUTs, such as Windows desktop, Java desktop, Web, or Android, requiring users to configure and maintain a diverse set of scriptless tools for evaluating quality across different types of systems;
2. Some tools have not been evaluated with diverse and complex industrial systems;
3. Many tools have not been maintained for years after their initial publication or are not available as open-source software.

1.4.1 TESTAR tool

TESTAR is an open-source scriptless testing tool that emerged between 2010 and 2013 as a result of the European research project Future Internet Testing (FITTEST) [280, 279]. TESTAR follows a modular plugin architecture, enabling the tool to extend the concept of state and action for various types of System Under Tests (SUT), integrate Action Selection Mechanisms (ASM) for making intelligent action selection decisions, and offer a flexible programmatic protocol for customizing different types of test oracles (see Figure 1.8).

Through collaborations with industrial partners such as Clavei [32], SOFTEAM [36], Berner & Mattner company [277], and Capgemini and ProRail companies [54], TESTAR has evolved significantly with various plugin extensions and has already demonstrated the capability to complement industrial testing approaches and uncover unexpected failures in industrial systems.

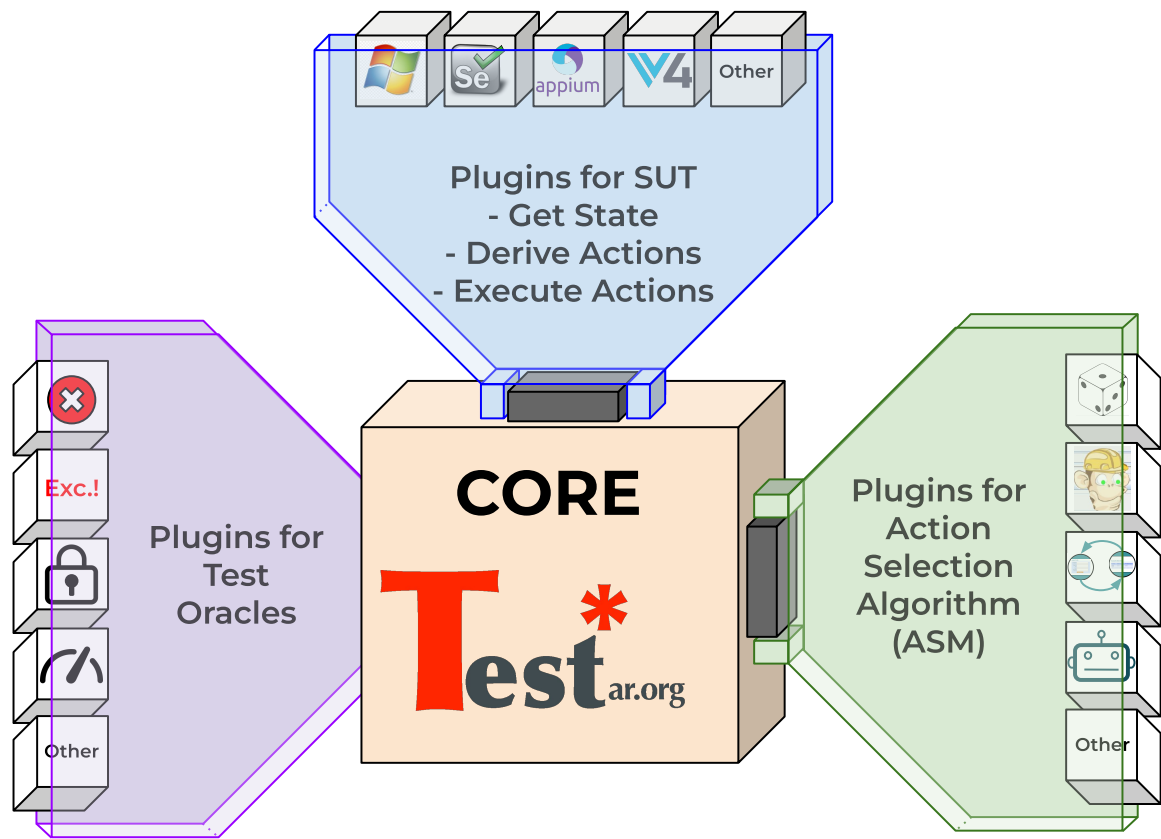


Fig. 1.8 TESTAR modular architecture overview

The initial version of TESTAR integrated the macOS Accessibility API to support the GUI testing of native macOS desktop and web applications [33]. Shortly after that initial version and resulting from an industry-academia collaboration, the tool was extended with the Microsoft UI Automation Accessibility API, enabling scriptless testing of native Windows desktop applications [32]. Additionally, a study with a smart home system demonstrated the modular capability of integrating non-GUI APIs, such as a REST API, allowing TESTAR to automate the testing of various Internet of Things (IoT) devices [173].

The default ASM of TESTAR relies on a purely random selection. However, in its early versions, the tool integrated a Reinforcement Learning (RL) ASM technique known as Q-Learning [34]. The logic programming language Prolog was also integrated to derive sensible and sophisticated actions, selected either randomly or through the Q-Learning algorithm [35]. Moreover, a Genetic Programming (GP) approach, was integrated and studied in TESTAR to evolve action selection rules [81, 82].

The initial versions of TESTAR supported test oracles designed to detect unexpected SUT crashes, unresponsiveness due to freezes, and suspicious exception and error messages using regular expressions [33, 36, 32, 277]. Furthermore, industry-academia collaborations

led to the development of diverse and system-specific oracles, such as detecting layout display failures [32], localization word issues [172], and IoT communication faults [173]. Finally, state model inference was employed to apply offline oracles for testing accessibility properties using the WCAG2ICT guidelines [145, 73].

TESTAR challenges

Despite the benefits of the TESTAR in complementing industrial testing approaches and uncovering unexpected failures, there are challenges that affect the efficacy and efficiency of this scriptless testing tool for complex systems:

1. TESTAR requires the integration of advanced technological frameworks to facilitate scriptless testing interaction with dynamic web and mobile systems, which were previously identified as constraints in prior studies [246, 13]. Additionally, research the extension of the tool with the capability to connect and interact with complex eXtended Reality (XR) systems.
2. TESTAR requires to continue the research of applying new exploratory ASMs and extend the graph-based memory while considering abstraction challenges to enhance exploration effectiveness and efficiency [246, 73]. Moreover, complex XR systems will require the development of new decision-making algorithms tailored to navigate and explore 3D environments.
3. TESTAR requires the definition of relevant data inputs, configuration actions, and test oracles to cover a more exhaustive search space and detect failures in the SUT. Additionally, appropriate test adequacy criteria must be integrated to determine whether scriptless testing complements the manual or scripted testing approaches used in the industry. Furthermore, it is relevant to continue the applicability of inferred graphs as offline oracles [73], enabling the detection of delta changes.
4. TESTAR requires evaluating the effectiveness and efficiency of its novel scriptless techniques in complex software systems, including industrial applications and well-established open-source applications with broad user communities. This also involves establishing a TESTAR development environment for verifying and validating tool changes, as well as integrating it into continuous integration environments.

1.5 Research Goal

The long-term objective of scriptless testing is to achieve a paradigm shift in software testing: *from developing test scripts to developing intelligent scriptless agents*. Our research group envisions establishing a core testing framework that employs robust technological capabilities and AI-driven adaptable agents capable of assessing various software quality characteristics. Achieving this vision requires continuous progress in identifying, addressing, evaluating, and disseminating the challenges that scriptless testing tools encounter when testing complex software systems.

The main research goal of this thesis is to **Improve the Quality Assurance testing process of desktop, web, mobile, and eXtended Reality (XR) applications through scriptless testing**. This thesis contributes to advancing the scriptless testing paradigm shift by addressing challenges identified with TESTAR, a scriptless testing tool that emerged from the FITTEST project and has been under development and research by the Universitat Politècnica de València, Spain, since 2010.

1.5.1 Objectives

We established four objectives to achieve the main research goal that focuses on addressing the TESTAR scriptless challenges (see Figure 1.9):

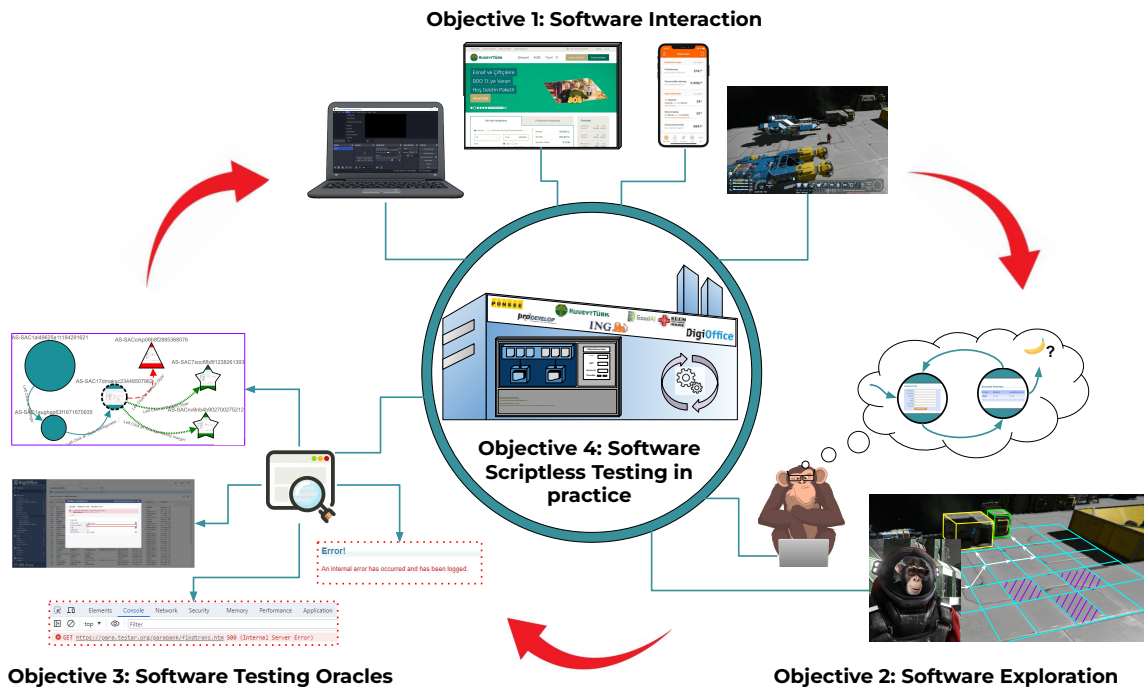


Fig. 1.9 TESTAR scriptless thesis objectives

- **Objective 1 (Software Interaction):** Obtain information on 2D widgets from desktop, web, and mobile systems, as well as 3D entities in XR environments, by integrating existing or innovative accessibility APIs and automation frameworks into TESTAR.
- **Objective 2 (Software Exploration):** Improve the action derivation, model inference for memory, and abstraction mechanism by adapting TESTAR to the characteristics of the SUT. On top of this robust basis, implementing various intelligent decision-making algorithms for action selection can potentially improve exploration and navigation effectiveness. Moreover, distributed approaches that share memory can reduce exploration time and improve scriptless efficiency. Finally, to empirically evaluate the effectiveness and efficiency of these exploratory techniques, it is necessary to apply test adequacy criteria to measure the test coverage of the SUT.
- **Objective 3 (Software Testing Oracles):** Integrate different types of oracles to automatically identify diverse software failures, including functionality, usability, security violations, and other invariant system characteristics. These test oracles can be integrated to check the system at runtime or be used after the exploration process using model-inferred information.
- **Objective 4 (Software Scriptless Testing in practice):** Evaluate the enhanced TESTAR scriptless capabilities by elaborating empirical studies with industrial and well-established open-source applications. Experiments must be conducted across various desktop, web, mobile, and XR SUTs. The objective is to ascertain whether the automated scriptless technique improves the testing process by complementing existing coverage or uncovering unexpected bugs.

1.5.2 Methodology

To achieve the four established objectives and the main thesis goal, we collaborate with industrial partners through EU projects to research, identify, and implement solutions for real industrial challenges. The research methodology of this thesis aligns with the design science paradigm [130].

We delineate the following methodological phases:

1. **Problem identification:** Evaluate the technological challenges companies encounter in their software testing practices with complex applications and analyze the relevance of designing processes to ensure the quality of their products. To achieve this, we collaborate and exchange knowledge with the universities and companies that participate

in the European projects TESTOMAT, DECODER, iv4XR, IVVES, ENACTEST, and AUTOLINK as well as with other external partners interested in the TESTAR tool.

2. **Solution research, design, and development:** Investigate existing automated scriptless processes intended to maintain the software quality to propose solutions that complement manual or script-based approaches. Subsequently, design and develop these automated scriptless solutions by enhancing an existing artifact, the TESTAR tool, to achieve the thesis objectives.
3. **Empirical Evaluation:** Validate the proposed and designed solutions developed on the TESTAR artifact, particularly regarding coverage effectiveness, bug detection effectiveness, and exploration efficiency. For this evaluation, we adhere to a software testing methodological framework that guides the definition and measurement of effectiveness and efficiency metrics within a design science methodology [278]. Depending on the SUT requirements and software testing experiment objectives, we establish suitable experimental test environments, collect and analyze data based on predefined criteria, and statistically analyze the data to validate our research objectives.
4. **Communication of research:** Document the entire research process following scientific dissemination open-source guidelines to finally obtain and present a comprehensive set of conclusions to the scientific research community. Since a research thesis is an evolving process of analysis and learning, new areas for future research and challenges related to scriptless testing have emerged. These future directions are detailed in the discussion section.

1.6 Context

This thesis has been produced within the Research Center on Software Production Methods (PROS) and the Valencian Research Institute for Artificial Intelligence (VRAIN) research group at the Universitat Politècnica de València (UPV) in Spain.

The author of this thesis started a bachelor project at UPV in 2017 by integrating the Java Access Bridge technology in TESTAR to enable the tool to connect and test Java Swing desktop applications. After the bachelor project in 2018, the author of this thesis continued as a technical researcher at UPV, collaborating with research partners from the Open Universiteit (OU) of the Netherlands. In the first years of technical research at the UPV, the author of this thesis started to collaborate with academic students and industrial partners from European projects to extend the capabilities of TESTAR, solve some of the

tool's challenges, and publish the results of these studies in research conferences. Then, in mid-2021, he officially joined as PhD student to begin leading the publication of research studies while still collaborating with academic and industrial partners.

The following competitive European research projects funded the knowledge obtained and generated during these years:

- ITEA3 TESTOMAT project (TESTOMAT 16032) from Oct 2017 to Dec 2020
- H2020 DEveloper COmpanion for Documented and annotatEd code Reference (DECODER 824231) from Jan 2019 to Dec 2021
- H2020 Intelligent Verification/Validation for Extended Reality Based Systems (IV4XR 856716) from Oct 2019 to Dec 2022
- ITEA3 Industrial-grade Verification and Validation of Evolving Systems (IVVES 18022) from Oct 2019 to Jun 2023
- Erasmus+ European Innovation Alliance for Testing Education (ENACTEST 101055874) from Sep 2022 to Aug 2025
- NWO OTP project Automated Unobtrusive Techniques for LINKing requirements and testing in agile software development (AUTOLINK 19521) from 2023 to 2026

1.7 Thesis Structure

The thesis contains nine chapters in addition to this introduction. Six of these chapters correspond to the established TESTAR scriptless testing objectives and include their respective empirical evaluations. An additional chapter presents an industrial experience study that extends scriptless testing to apply Behavior-Driven Development (BDD) for automating regression testing. The thesis concludes with two chapters: one discussing the achievement of the main thesis goals and the learning process, and the final chapter outlining conclusions and future work directions.

- **Chapter 2. TESTAR: Software interaction.** In this chapter, we first introduce how accessibility APIs and technical automation frameworks enable the TESTAR tool to identify the existing range of widgets in GUI or XR systems that constitute the state of the SUT. Secondly, we explain the various types of actions that the TESTAR tool can execute to interact with the SUT. Finally, we indicate the environments used for GUI software interaction.

- **Chapter 3. TESTAR: Software exploration.** In this chapter, we delve into the testing objectives of employing diverse ASMs when exploring the SUT in a scriptless automated approach. We emphasize the need to structure decision-making processes over a reliable constitution of widgets, states, and action identifiers. Additionally, we underscore the benefits of maintaining memory regarding the parts discovered and pending exploration. Subsequently, we detail novel ASMs developed and integrated into TESTAR to enhance exploration effectiveness and efficiency for GUI and XR systems.
- **Chapter 4. TESTAR: Distributed exploration.** In this chapter, we present a distributed approach that enables multiple scriptless testing agents to coordinate in achieving a common goal. Multiple TESTAR instances share centralized SUT knowledge, allowing them to coordinate their actions and improve the efficiency of exploring GUI systems.
- **Chapter 5. TESTAR: Online test oracles.** In this chapter, we describe diverse types of software failures that the scriptless exploratory approach can detect at runtime during the exploration of the SUT. Moreover, we enumerate potential online test oracles and actionable strategies to identify these failures, along with the challenges involved in integrating them into the TESTAR tool.
- **Chapter 6. TESTAR: Offline delta testing oracles.** In this chapter, we emphasize the importance of being aware of delta changes and provide a systematic mapping of the literature to study existing techniques for detecting delta changes in GUI systems. We then introduce an offline delta testing tool that uses GUI state models, inferred by TESTAR, to automatically detect and highlight GUI delta changes between two versions of the same SUT.
- **Chapter 7. TESTAR: Scriptless testing in practice.** In this chapter, we share our industrial experiences from collaborating with industry partners to validate the capabilities of TESTAR in diverse real-world environments. The feedback gathered through these collaborations has been instrumental in refining TESTAR, enhancing its effectiveness as a scriptless testing solution tailored to meet industrial requirements.
- **Chapter 8. Beyond TESTAR: A BDD testing approach.** In this chapter, we extend our testing research beyond the TESTAR scriptless approach. Collaborating with an industrial game company showcased the importance of integrating intelligent agents capable of following natural language instructions for testing game functionalities

related to the company's regression scenarios. To achieve this, we implemented a Behavior-Driven Development (BDD) agent for game testing.

- **Chapter 9. Discussion.** In this chapter, we discuss and reflect on the overall achievements of this thesis concerning the planned objectives and their potential contributions to the research community, industry, and other stakeholders. Additionally, we argue about emerging challenges and future research areas in the scriptless testing field.
- **Chapter 10. Conclusions and future work.** In this concluding chapter, we summarize the work of this thesis, reflecting on the objectives accomplished and the challenges encountered. We discuss the achievements and implications of our findings within the context of software testing. Finally, we describe future lines of research and promising opportunities for continuing to advance in the scriptless testing realm.

Chapter 2

TESTAR: Software interaction

Software applications utilize textual information together with visual 2D widgets, 3D entities, or sounds to display to users the desired information for learning, management, entertainment, or other software purposes. While certain displayed text, 2D widgets, or 3D entities are solely dedicated to providing information to the users, software applications also contain interactive interfaces, widgets, or entities on which users can perform actions through input devices (e.g., computer mouse, keyboard, etc.) to interact with the software. This interaction can involve writing instructions via Command Line Interfaces (CLI), capturing event and gesture actions over displayed 2D widgets or 3D entities, or recognizing voice commands.

For test automation tools to be capable of performing automated executions without the need for human intervention, these tools must technically be able to extract textual, visual, and/or sound information and execute interactions. This extracted information about the Software Under Test (SUT) can be used to derive or determine the possible range of interactions to perform within the software (e.g., commands and syntax available in a CLI or click mouse events over a 2D button). Furthermore, this information is used to apply test oracle conditions that determine if the SUT contains failure (e.g., an internal software variable points to a null reference and crashes).

In today's software systems, most desktop, web, and mobile applications provide users with a Graphical User Interface (GUI) for displaying textual and visual information on a 2D screen. This GUI comprises interactive 2D widgets where users can mainly execute write, click, and drag actions to interact with the SUT. Moreover, some of these systems also provide information about the internals of the software and allow CLI or console instructions for interaction. Section 2.1 details how the TESTAR tool obtains the GUI screen information, derives the possible action to execute in the GUI, and additionally reads the SUT process buffers or automation framework events to interact via CLI or console instructions.

Novel eXtended Reality (XR) software applications allow users to immerse themselves in 3D environments that contain virtual entities that display textual and visual information but also simulate real-world physics. Unlike GUI applications, in complex 3D environments, obtaining the software information based on visible 2D entities on the screen or executing basic actions like clicks or keyboard inputs is insufficient for reliable test automation. These environments require the observation of surrounding 3D entities, as well as notions of 3D orientation and movements, to interact with XR SUTs. Section 2.2 delves into how the TESTAR agent observes the surrounding entities' information in a 3D environment and derives the possible actions from executing in the 3D environment. Finally, Section 2.3 summarizes both GUI and game TESTAR software interaction.

2.1 Interaction with Desktop, Web, and Mobile systems

The modular architecture of the TESTAR tool facilitates the integration of various accessibility APIs, accessibility frameworks, automation frameworks, object recognition techniques, and other technical interfaces as system connection plugins. These plugins enable TESTAR to connect, extract the GUI information, and interact with a diverse range of software systems. In previous studies, TESTAR was able to interact with MacOS desktop and web applications using the MacOSX Accessibility API [298, 35], with Windows desktop and web applications using the Windows UI Automation Accessibility framework [114, 32], and demonstrated the tool adaptability by interacting with Internet Of Things (IoT) devices using RESTful API web services [236, 173].

While these technical plugins enabled TESTAR to interact with a wide range of desktop and web applications, the tool had certain limitations. For instance, Java Swing applications pose a challenge as they create GUI components at the Java Virtual Machine level rather than at the operating system level, preventing TESTAR interaction [212]. Moreover, while Accessibility APIs facilitate interaction with web applications through native desktop browsers (i.e., Safari for MacOS and Internet Explorer for Windows), this approach entailed limitations for dynamic applications or non-native and well-adopted browsers like Google Chrome [13]. Additionally, the widespread adoption of mobile applications and the necessity of testing the millions of emerging applications made it valuable to integrate plugins capable of interacting with mobile software applications [246, 144].

In this thesis, we tackled these limitations by integrating new technical plugins in the TESTAR tool to interact with desktop, web, and mobile systems [276, 128]. Java Access Bridge technology supports the interaction with Java Swing desktop applications [274, 189]. Selenium WebDriver automation framework enhances interaction with dynamic environments

of modern web applications [5, 240, 122]. The Appium WebDriver automation tool empowers TESTAR to interact with mobile ecosystems, encompassing Android and iOS platforms [128]. Furthermore, we have continued maintaining the Windows UI Automation Accessibility framework by augmenting support for an extensive array of automation properties [189].

In the following subsections, we describe how TESTAR obtains the GUI state (see subsection 2.1.1), how derives the set of possible actions to execute in the GUI (see subsection 2.1.2), and the virtual environments used for GUI interaction (see subsection 2.1.3).

2.1.1 Obtaining the Graphical User Interface state

A Graphical User Interface (GUI) is composed of a set of *widgets*. These widgets are organized in a hierarchical parent-child structure called the *widget tree*. Each widget contains a set of properties that represent its various textual, visual, and functional characteristics.

To obtain the GUI widget tree information, we rely on various technical plugins (i.e., MacOSX Accessibility API, Windows UI Automation Accessibility framework, Java Access Bridge, Selenium WebDriver, and Appium WebDriver). Due to the diverse characteristics of desktop, web, and mobile GUI system environments and the underlying technology of software applications, each plugin exposes widget properties using distinctive names and identifiers. Still, these properties exhibit GUI similarities.

Given that GUI widgets are rendered on a 2D screen, they possess properties that represent their **position** and **size**. Moreover, widgets display information as a textual **title**, provide accessibility **helpText** descriptions, or contain internal **identifiers**. Each widget is categorized into a type of **role**, such as button, text field, checkbox, dropdown, and more. Various technical properties can be employed to determine common behavior, such as determining if the widget is **enabled** for user interactions or can be **focused** by mouse or keyboard events.

Additionally, it is possible to customize widget properties that employ or combine original properties exposed by the technical plugin. For instance, in the context of Windows desktop applications, we can customize and deduce whether a widget is **blocked** or not depending on if the widget belongs to a **modal** dialog window that is **readyForUserInteraction**. Conversely, in web applications, we can iterate through web elements and HTML iframes to check obscured widgets to determine this custom **blocked** property. Furthermore, using the widget's position in the widget tree, we can calculate and define the notion of numerical **path** or **xpath** predicate expression.

The set of all widgets from the *widget tree* and their properties constitute the GUI *state* s . The nodes of the *widget tree* are the widgets rendered on the GUI in a particular state s . We denote this set of nodes with $W(s) = \{w_1, w_2, \dots, w_k\}$ (for example, button, slider, text

field, menu, etc.). The edges of the tree reflect the parent-child relationships: each child widget is displayed within the screen area occupied by its parent widget. We denote the set of edges with $E(s)$. Consequently, there exists a directed edge $(w_i, w_j) \in E(s)$ when $w_i \in W(s)$ is the parent widget of $w_j \in W(s)$ in state s . The values of all the widgets' properties also define the state. For a widget $w \in W(s)$, we use $P(w, s)$ to denote the set of all properties $\{w.p_1, w.p_2, \dots, w.p_m\}$ (for example **role**, **title**, **position**, **enabled**, etc.) for that widget w in state s . The GUI states from Figure 2.1, Figure 2.2, and Figure 2.3 represent how different desktop, web, and mobile software applications can be similarly represented as widget trees.

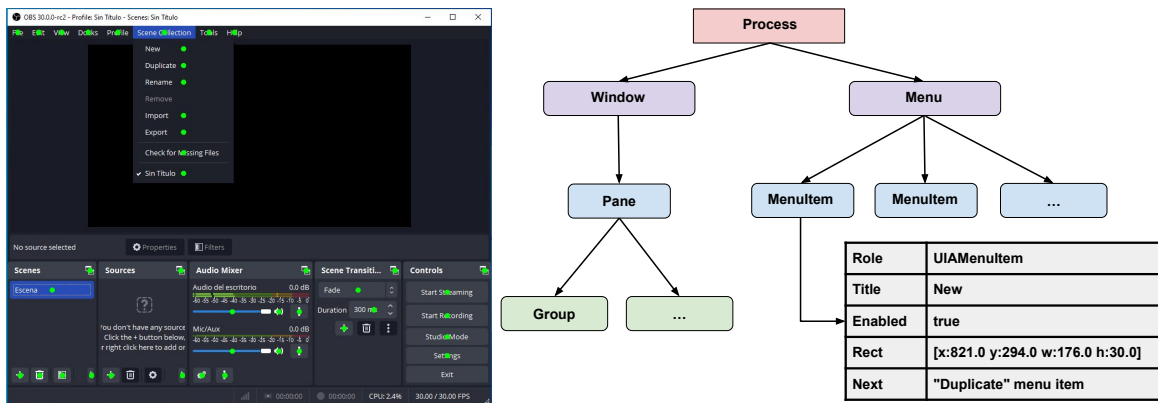


Fig. 2.1 The widget tree of a Desktop application that constitutes the GUI state

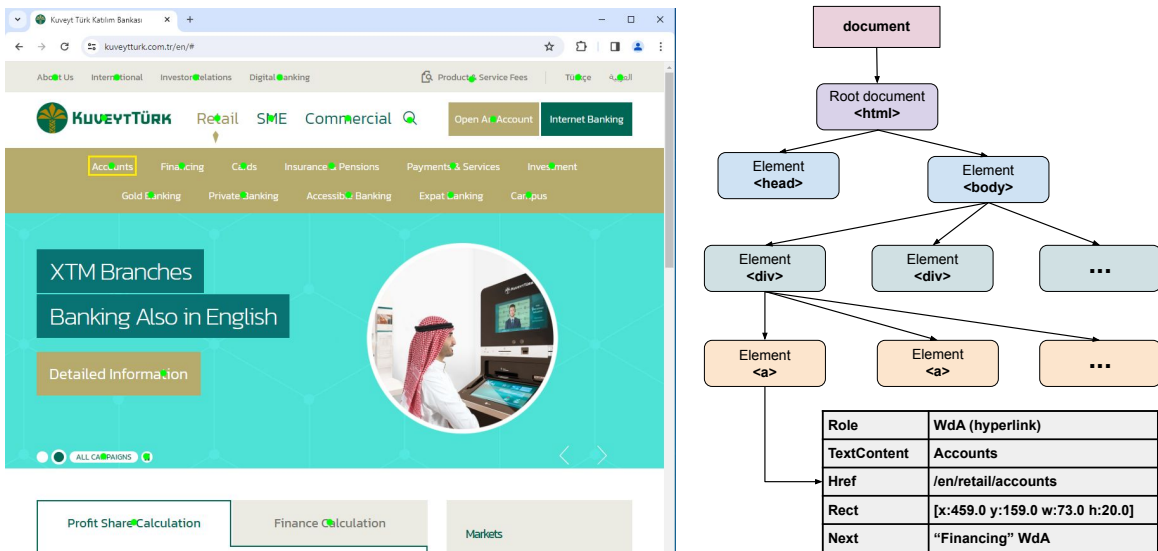


Fig. 2.2 The widget tree of a Web application that constitutes the GUI state

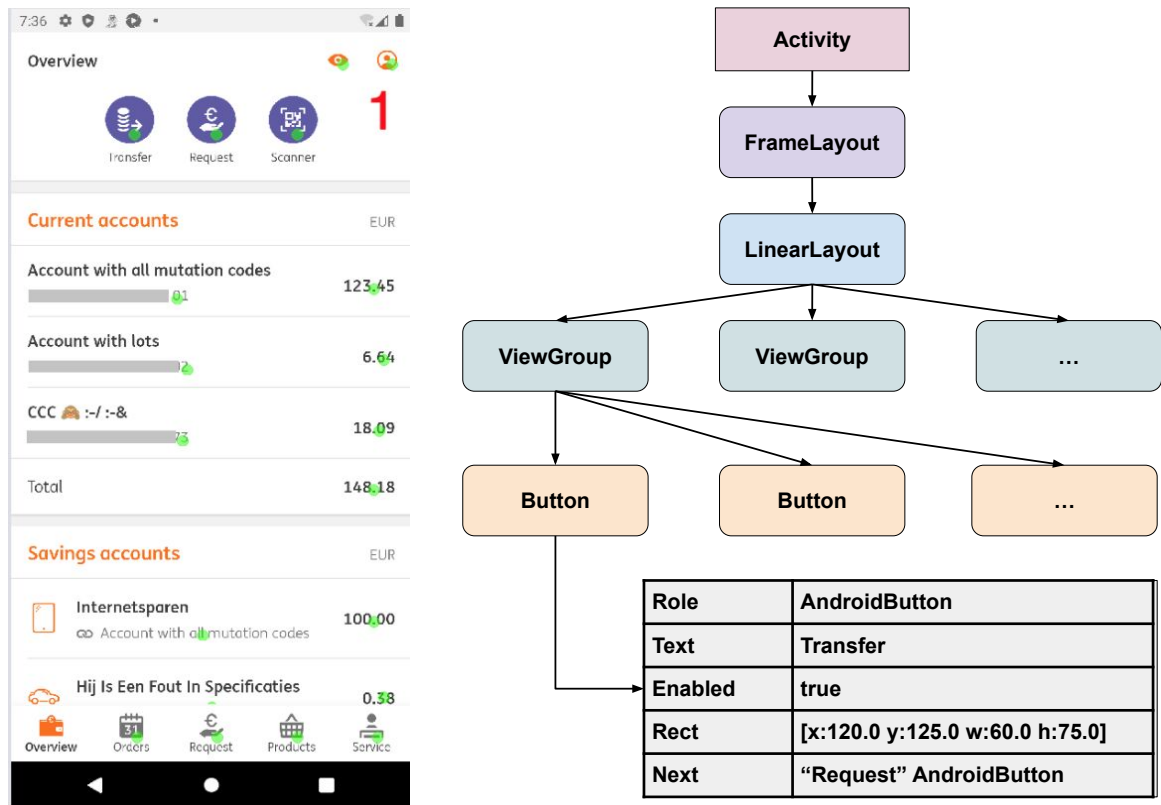


Fig. 2.3 The widget tree of a Mobile (Android) application that constitutes the GUI state

All the properties $P(w, s)$ obtained in state s for the widgets in $W(s)$ through an API or an automation framework are associated with the TESTAR representation of Widgets, States, and Actions. This association is done through Tags and is depicted in Figure 2.4.

Taggable classes implement the Taggable interface, which means that Tags can be added to their instances. In TESTAR, the classes `Widget`, `State`, and `Action` are taggable, and the Tags are pairs of (property name, value). Properties that are common to all widgets are defined in a final class `Tags`. The properties that are specific to an implemented API technology or automation framework (like Windows UIAutomation, Selenium WebDriver, Appium WebDriver, etc...) are defined in specific API-Taggable final classes (`UIATags`, `WdTags`, `AndroidTags`, `IOSTags`, etc...). We can use the `set` method to attach or update a property value of a taggable object and the `get` method to read the value of the properties of a taggable object (see Example 2.1).

Extending the SUT state information

In Windows systems, it is possible to get the process identifiers (i.e., unique process or multi-process identifiers of software applications [11, 237]) of the SUT and the handle of the

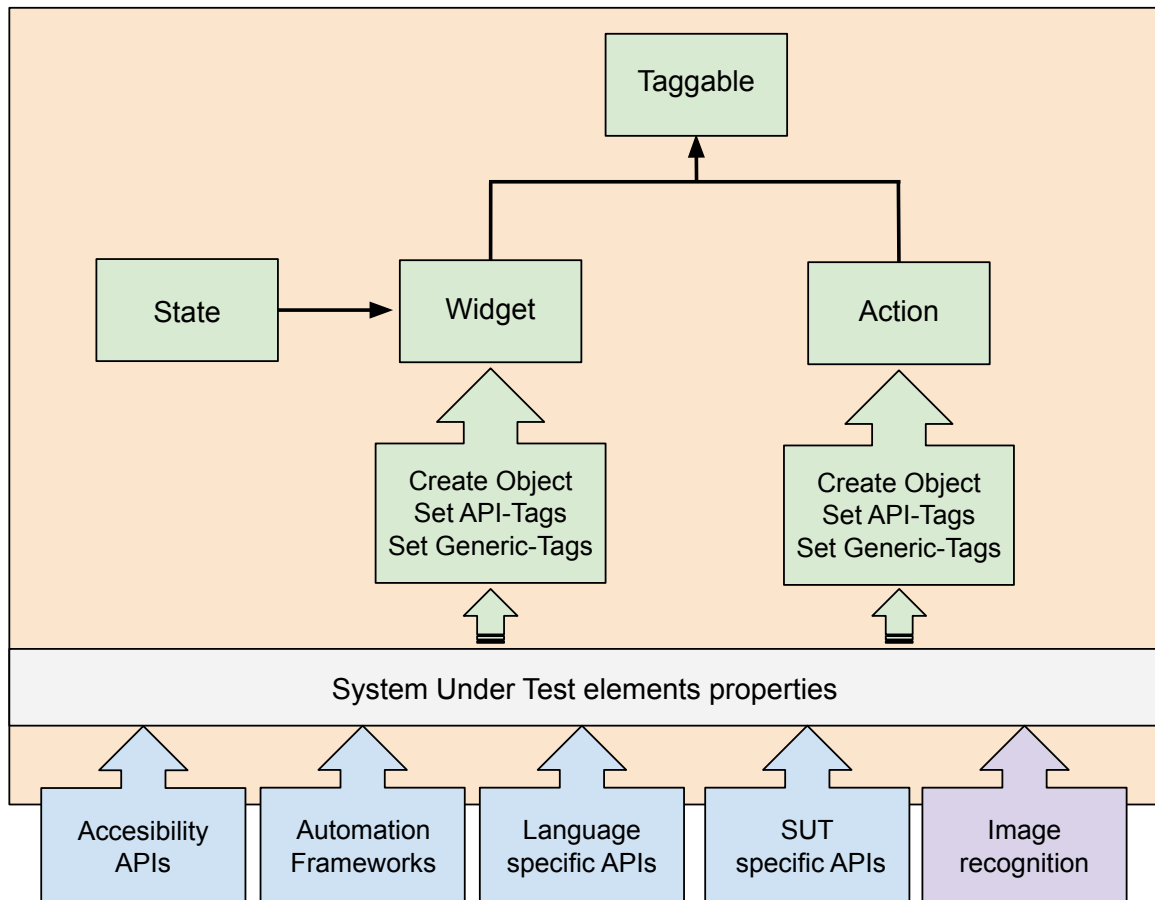


Fig. 2.4 Taggable Widget, State, and Action classes obtained from a System Under Test

```

taggableObjectName . set ( Tags . PropertyName , PropertyValue )
taggableObjectName . get ( Tags . PropertyName )
  
```

Example 2.1 TESTAR Taggable methods

windows that compose the SUT state (i.e., a SUT process can have zero, one, or multiple GUI windows). In this way, we can use the Windows API to determine if the handle of the foreground window matches with some of the windows handles of the SUT. Moreover, TESTAR can use the SUT process identifier for diverse purposes, such as monitoring the CPU and memory usage of the SUT in each state or connecting with the output and error stream of the process to obtain SUT debugging information.

In Web systems, the Selenium WebDriver automation framework offers the possibility of executing JavaScript commands to obtain web page data. For example, to check the `document.readyState` JavaScript property that determines if the web page state has been loaded. Furthermore, the Selenium DevTools API provides capabilities to inspect and extract

data from the browser's network interface. Additionally, Selenium WebDriver facilitates managing control and switching navigation between browser windows tabs.

Regardless of the type of system, logging is a fundamental practice in software development. This practice involves generating log files that capture internal system operations. These logs provide information that helps debug and analyze software execution to identify potential internal failures that may not be visible in the GUI.

For Windows systems, TESTAR incorporates functionality enabling the connection to and extraction of data from local log files. In web systems, the Selenium Log API provides the capability to monitor web console events. Similarly, in mobile systems, Appium WebDriver facilitates the capture of device logs during testing.

2.1.2 Deriving Graphical User Interface actions

Having obtained the GUI state information, TESTAR derives a set of GUI actions that aim to simulate the available interactions a user can perform in a specific state. TESTAR uses the **rect** property, which indicates the widget's position within the GUI, to make **movement actions** over the widget. Subsequently, it primarily uses the **role** property of each widget to determine whether it belongs to a native interactive element and define what type of action to derive:

- **Click actions:** applies to widgets that users consider clickable, including buttons, menu items, checkboxes, input-submit, hyperlinks, and so on, depending on the type of software system. Moreover, it is possible to distinguish between left and right click or derive a double click action.
- **Type actions:** applies to widgets that users consider typeable, including edit-text fields, text areas, input-text, and so on, depending on the type of software system.
- **Drag actions:** applies to widgets that users consider draggable, including sliders, scroll bars, and so on, depending on the type of software system.

The TESTAR tool does not derive any action by default in widgets like panels, labels, divs, spans, etc., mainly considered containers to group child widgets, prepare aesthetics, or show textual information without expecting an interaction. Nevertheless, some software applications create new non-native GUI elements or customize default non-interactable native elements to react to interactions. For these cases, TESTAR offers configurable settings and a programmatical protocol that allows users to adapt the action derivation behavior to the needs of their SUT.

Listing 2.2 demonstrates how TESTAR can be configured to derive a click action by reading the **class** property of a web widget when it corresponds to the Angular “mat-icon” component. Similarly, as shown in Listing 2.3, TESTAR can derive a type action if the **class** property matches the Angular “mat-form-field” component.

```
1 @Override
2 protected boolean isClickable(Widget widget) {
3     if (widget.get(WdTags.WebCssClasses).equals("mat-icon")){
4         return true;
5     }
6     return super.isClickable(widget);
7 }
```

Listing 2.2 TESTAR protocol to configure "mat-icon" Angular widgets as clickable

```
1 @Override
2 protected boolean isTypeable(Widget widget) {
3     if (widget.get(WdTags.WebCssClasses).equals("mat-form-field")){
4         return true;
5     }
6     return super.isTypeable(widget);
7 }
```

Listing 2.3 TESTAR protocol to configure "mat-form-field" Angular widgets as typeable

On certain occasions, there are widgets on which it is not suitable to derive any action, even if this widget is considered interactive. For instance, it might not make sense to interact with disabled widgets (i.e., the property **enabled** is false), widgets that are **blocked**, or widgets that are rendered in the GUI but that are not visible in a specific state (e.g., a web application may contain loaded interactive widgets that are below the visible canvas of the browser). The previous Figure 2.1, shows how the menu item with the title “Remove” is not deriving a click action (i.e., a green dot indicates when TESTAR derives a click action) because the widget is not **enabled**.

Furthermore, even in cases where a widget is interactive, visible, and enabled to interact, it might be necessary to avoid deriving an action on it during the testing process. For example, when testing an application, we probably want to avoid clicking on the “Close”, “Exit”, or “Logout” buttons, which potentially will close the SUT and interrupt the scriptless explorative process. For these cases, the TESTAR tools allow users to manually interact with the SUT to define a *blocklist* list of undesired specific widgets and to use regular expressions to define a *filtering* list for groups of undesired widgets. In Figure 2.1, it is possible to appreciate that

the “Minimize”, “Maximize”, and “Close” widgets that exist in the top-right SUT corner are not deriving a click action due to TESTAR computing the regular expression to filter actions (see Example 2.4).

```
ActionFilter = .*[mM] aximi [ zs ] e . * | .* [ mM ] inimi [ zs ] e . * | .* [ cC ] lose . *
```

Example 2.4 TESTAR regular expression to filter non-desired actions

The Figure 2.5 represents TESTAR’s internal logic sequence for determining whether to derive an action on a widget and what type of action to derive.

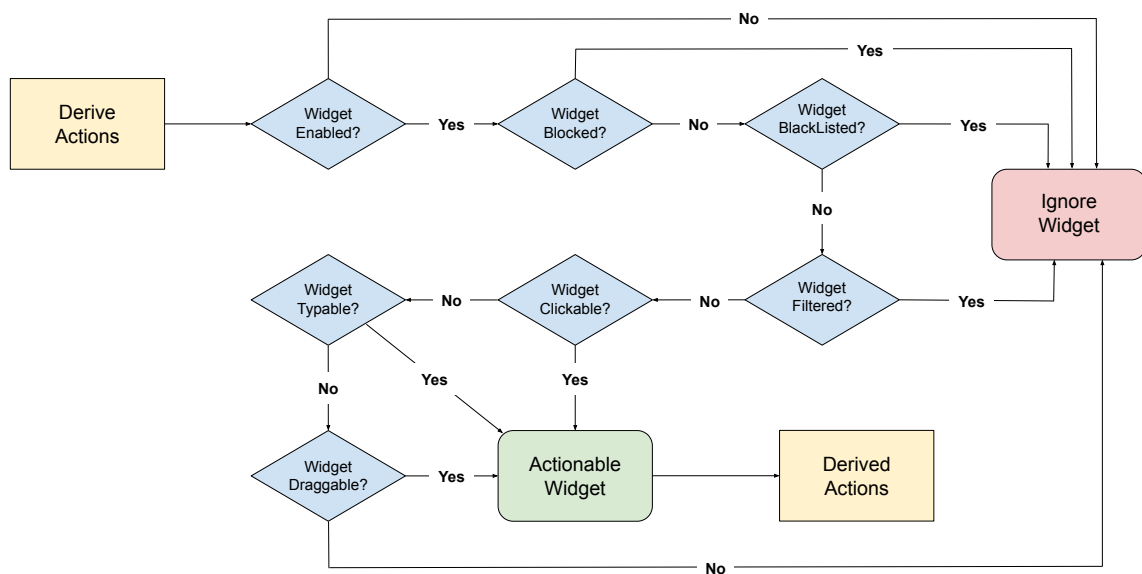


Fig. 2.5 Diagram of action derivation logic for interactive widgets

GUI action events

To perform the click, type, and drag GUI actions, TESTAR is required to generate a set of native system input events. For instance, typing text into an edit-text field GUI widget requires several event steps:

1. Determine the position of the widget on the screen;
2. Move the mouse cursor to that position;
3. Press the mouse down;
4. Release the mouse to enable typing; and
5. Type the desired text by pressing and releasing the char keyboard key codes;

Each of these action events has its own `Action` implementation: `MouseMove`, `MouseDown`, and `MouseUp` for cursor movement and mouse button events; `Type` to abstractly represents the typed text and `KeyDown` and `KeyUp` for low-level keyboard input events. Additionally, because the notion of `Action` is a compound of events, TESTAR implements the `CompoundAction` class to aggregate sequences of events into an `Action`. By default, TESTAR relies on the Java AWT Robot library to generate these native system input events ¹.

TESTAR uses, by default, an `InputDataManager` class to randomly compute a number, generate alphanumeric text, or select a URL, email, or date from a predefined set of valid and invalid values. However, a typing action must take into account the context of the SUT scenario, the objective of the testing procedure, or additional information, such as the desired length of the generated text. For instance, preparing a scriptless test execution to check combinations of valid and invalid alphanumeric characters and dates for a bank transfer feature is different from crafting a large set of security injection payloads to test potential vulnerabilities in the bank SUT. Therefore, TESTAR also allows users to use custom text files as input lists for deriving typing actions.

As an example of configuring a TESTAR protocol for deriving type actions, Listing 2.5 illustrates how to iterate through each enabled, non-blocked, and non-filtered typeable widget in the state, use the `InputDataManager` class to retrieve a random text from a specified file, and derive a type action. Listing 2.6 shows how to create a type action—first, using mouse events to move and click the position of the desired widget and, secondly, invoking the `Type` class. Then, Listing 2.7 presents how the `Type` class internally iterates through all the text chars to perform keyboard input events.

Enriching SUT action interactions

In Windows systems, it is also possible to derive control actions to force the SUT in the foreground or kill undesired processes. When the Windows API detects that the handle of the foreground window does not match any window handles of the SUT, it is possible to use native calls intended to set the main window to the foreground or combine keyboard events such as “Alt + Tab”. To kill undesired processes, it is feasible to constantly check the existing processes in our SUT environment after each action.

For Web systems, as mentioned previously, the Selenium WebDriver automation framework offers the possibility of executing JavaScript commands to interact with the browser window and with the web elements that exist in the current web document (i.e., calling `WebDriver.executeScript(JScommand)`). TESTAR predefines a couple of JavaScript commands to define actions that are considered useful during testing.

¹<https://docs.oracle.com/javase/8/docs/api/java/awt/Robot.html>

```

1 protected Set<Action> deriveActions(State state){
2     // Derive actions for the current state
3     Set<Action> actions = new HashSet<Action>();
4     // Iterate through each widget in the state
5     for (Widget widget : state) {
6         // If the widget
7         if (widget.get(Tags.Enabled) // is enabled
8         && !widget.get(Tags.Blocked) // is non-blocked
9         && !widget.get(Tags.Filtered) // is non-filtered
10        && isTypeable(widget)){ // and is typeable
11            // Get a random text from a specified file
12            String text = InputDataManager.getTextFromFile(file);
13            // To derive a type action
14            Action typeAction = clickTypeInto(widget, text);
15            actions.add(typeAction);
16        }
17    }
18 }

```

Listing 2.5 TESTAR protocol to derive type actions from the widgets of the state

```

1 public Action clickTypeInto(Widget widget, String text){
2     // Obtain the widget position by getting the rectangle Shape
3     Position position = widget.get(Tags.Shape);
4     return new CompoundAction.Builder()
5         .add(new MouseMove(position))
6         .add(MouseDown)
7         .add(MouseUp)
8         .add(new Type(text))
9         .build();
10 }

```

Listing 2.6 Compound typing action aggregating a sequence of mouse and type events

When testing web applications, in addition to deriving control actions to maintain the browser in the system foreground, it is necessary to ensure that the focus of the URL domain of the SUT is not lost and avoid exploring non-desired web pages. For this objective, TESTAR derives the JavaScript command actions:

- `WdCloseTabAction` to close a tab.
- `WdHistoryBackAction` to simulate a click on the history back button in a browser.

Moreover, JavaScript command actions allow interaction with the Document Object Model (DOM) API to find and manipulate web elements with their attributes. These web

```
1 public final class Type extends TaggableBase implements Action {
2     private final String text;
3     public Type(String text){
4         this.text = text;
5     }
6     public void run(){
7         // Iterate through the chars of the text
8         for(int i = 0; i < text.length(); i++){
9             char c = text.charAt(i);
10            KBKeys key = getKey(c);
11            // To perform keyboard input events
12            new KeyDown(key).run();
13            new KeyUp(key).run();
14        }
15    }
16 }
```

Listing 2.7 The Type class internally performs keyboard input events

event interactions may offer several advantages over traditional GUI actions regarding reliability and efficiency or flexibility and customization. For example, it can interact with large web forms, craft script queries for specific SUT events, interact with loaded but not visible widgets, or perform headless testing without a GUI environment.

Although the TESTAR protocol can be used for custom JavaScript commands, the tool also offers a set of predefined JavaScript actions:

- `WdSubmitAction` to simulate a click on a submit button in a detected web form.
- `WdAttributeAction` to find a web element by its unique identifier and write a value in the desired attribute using a pair (key, value).
- `WdSelectListAction` to select an element of a dropdown list (e.g., this is useful since items from select tag web elements have empty GUI dimensions).
- `WdUrlAction` to navigate to a new URL.
- `WdFillFormAction` to fill forms by reading XML templates and combining `Type` and `WdSubmitAction` interactions.
- `WdRemoteClickAction` to send a click interaction web event without using GUI events.
- `WdRemoteScrollClickAction` to scroll the desired widget into the visible area of the browser window before invoking `WdRemoteClickAction`.

- `WdRemoteTypeAction` to send a type interaction web event without using GUI events.
- `WdRemoteScrollTypeAction` to scroll the desired widget into the visible area of the browser window before invoking `WdRemoteScrollTypeAction`.

While for desktop and web applications, the SUT window or browser GUI can be displayed on the screen of the native operative system, mobile applications necessitate the emulation of Android or iOS environments. This emulation is achieved through sandbox devices like Android Studio, external virtual machines, or physical devices. Although this emulation allows for the possibility of executing mouse and keyboard GUI native system input events (if the emulator device is running as a desktop application), such methods lack efficiency.

As a result, when testing mobile applications, TESTAR does not perform any GUI interaction event. Instead, it uses the interfaces of the Appium WebDriver automation tool to send interaction events:

- `Click` for both Android and iOS environments.
- `Long-click` specifically for the Android environment.
- `Scroll` to simulate finger scrolling for both Android and iOS environments.
- `Type` to send sequences of key characters for both Android and iOS environments.
- `Back` to simulate a click on the back button only in the Android environment.
- `System` to execute tasks such as initiating a phone call, sending an SMS, or rotating the screen orientation, exclusive to the Android environment.

2.1.3 Virtual environments for GUI software interaction

In recent years, the use of virtual machines (VMs) to emulate computer environments has been a widely adopted practice for interacting with GUI software during the testing process [190, 9]. However, the industry is increasingly shifting towards containerization over traditional virtualization because it provides efficient, scalable, and cost-effective resource management solutions in company infrastructures [290].

A Docker container is a standalone unit of software that can package an environment, a SUT, and a GUI testing tool. For example, customizing a Docker container with OpenJDK enables the execution of a TESTAR tool Docker instance. Nevertheless, containerizing the desktop, web, or mobile environment is still necessary to achieve GUI software interaction.

In the context of web and mobile SUTs, containerization of the GUI environment is feasible using Docker containers. The Selenium community, for instance, provides an Ubuntu-based Docker image with a standalone version of Selenium ChromeDriver [252], allowing TESTAR to launch and interact with web SUTs. This Docker image also contains the *X virtual framebuffer* (Xvfb) [301] server that simulates graphical operations without the need to have any graphical or screen device in the system environment.

For mobile environments, emulators are necessary to simulate Android or iOS environments, enabling the execution of mobile GUI applications. Recent open-source projects, such as the budtmo project [47], offer Docker containers that deploy virtual mobile emulators alongside Appium as a remote service. TESTAR can then utilize this Appium service to obtain the GUI state information and send interaction events.

However, despite the availability of Docker containers for Windows, the containerization of Windows environments for GUI test automation remains an area requiring further research. This limitation arises because Windows containers do not support active GUI desktop sessions². Consequently, virtual machines are still the preferred solution for Windows desktop SUTs. These VMs must include a local instance of the SUT and a local instance of TESTAR, which enables the testing tool to connect and interact with the SUT within the virtual environment.

2.2 Interaction with computer 3D game systems

In the realm of traditional desktop, web, and mobile systems, technologies such as the MacOSX Accessibility API, Windows UI Automation Accessibility framework, Java Access Bridge, Selenium WebDriver, and Appium WebDriver, it is possible to enable the identification of GUI widgets and the subsequent interaction through native inputs like keyboard and mouse, or by sending interaction API events. However, game systems are not GUI applications in the traditional sense, as they do not expose a native widget tree. Instead, they are dynamic, interactive environments designed to immerse users in captivating virtual worlds that simulate 2D or 3D entities.

For traditional GUI applications, the aforementioned technologies suffice with the identification of GUI widgets that can be interacted with through keyboard and mouse inputs. However, these technologies fall short of meeting the necessary requirements for game systems. Game environments demand additional information to accurately identify their respective states, such as positional or orientation vectors for movements or properties associated with the objects being interacted with. Consequently, to employ scriptless testing for

²<https://github.com/microsoft/Windows-Containers/issues/47>

games, it becomes imperative to establish a connection between this testing approach and technologies capable of addressing the limitations of interacting with game environments:

- It is necessary to access the three-dimensional positions and orientations of game objects to determine if a game character can observe these objects or if they are obstructed. Additionally, it is important to obtain information about the variety of existing game objects and their properties to apply oracles for testing game functionalities.
- Depending on the type and properties of observed game objects, complex games may require a diverse set of interactions. Moreover, the logic to perform a game action typically involves combining multiple game interactions.
- Interacting with game objects in 3D scenarios requires navigating these scenarios to reach the desired game object while dealing with other potential obstructive objects.

To address these limitations, the European project Intelligent Verification/Validation for Extended Reality Based Systems (iv4XR) implemented a novel Java framework with a plugin architecture that provides a set of interfaces that can be implemented to connect, get information, and interact with virtual environments [227].

In this thesis, TESTAR was extended as a test *agent* capable of interacting with two 3D game systems by using two game plugins developed with the iv4XR framework [215, 226, 238]. This chapter presents the integration with the experimental 3D game LabRecruits, while Chapter 7.5 details the integration with the sandbox 3D game Space Engineers.

Subsection 2.2.1 introduces the iv4XR framework and the game plugin developed for the experimental 3D game LabRecruits. In the subsequent subsections, we describe how TESTAR can use the iv4XR capabilities to observe the entities that constitute the game environment state (see subsection 2.2.2), derive the set of possible actions to interact with game environments (see subsection 2.2.3), navigate 3D game environments (see subsection 2.2.4), and the virtual environments used for game interaction (see subsection 2.2.5).

2.2.1 iv4XR framework and game plugins

iv4XR is a Java framework with a plugin architecture that provides a set of interfaces that can be implemented to connect, get information, and interact with game objects. The *Entity* interface represents the existing game objects and their properties. The *Environment* interface allows connecting with game scenarios, observing the defined entities, and defining the actions that can be executed in the game. The *agent* can be any automated software testing tool that uses the environment to connect with the game and takes the role of a playable entity to observe the game entities, execute game actions, and apply test oracles.

The objective of iv4XR interfaces is to offer XR developers a seamless approach to create iv4XR-plugins that enable test *agents* (i.e., an automated testing tool or technique) to connect with the virtual environment of their XR system. These iv4XR-plugins should provide test *agents* a method to observe the virtual entities and action methods test *agents* can execute to interact with the virtual environment. The details of the iv4XR-game plugins for LabRecruits and SpaceEngineers are presented below.

LabRecruits game and the iv4XR plugin

LabRecruits³ is an experimental 3D game built in C# with Unity and constructed within the iv4XR framework to serve as a configurable game SUT to evaluate Artificial Intelligence (AI) test *agents*.

In LabRecruits, players can design dynamic labyrinths with interactive objects such as doors, buttons, fire, and goal flags. This experimental game has no specific goal but can simply use intelligent algorithms to explore the level or make the game character use AI decisions to reach some particular room after opening a door entity (e.g., open the door3 and reach the door3 position). Figure 2.6 shows a LabRecruits scenario with three button entities discovered and one unknown due to the walls, three door entities, one fire hazard entity, and the playable game character.



Fig. 2.6 LabRecruits game scenario

³<https://github.com/iv4xr-project/labrecruits>

The development of the iv4XR LabRecruits-plugin⁴ enables access to the internal data and functions of this 3D experimental game. In LabRecruits, the *agent* takes the role of the playable game character.

The LabRecruits-plugin consists of the game-server and client components. The server-side is integrated into the C# game code, which has a total of 5989 Lines Of Code (LOC) and allows the connection with the game by defining the properties and controller functions of game objects. The client-side is written in Java and implemented the interfaces defined of the Java iv4XR framework, has a size of 8263 LOCs, and provides classes that grant access to game data and logical functions like navigation for the *agents*. Figure 2.7 shows an overview of the plugin classes, which are used for observing the game state, deriving game actions, and navigating the game environment.

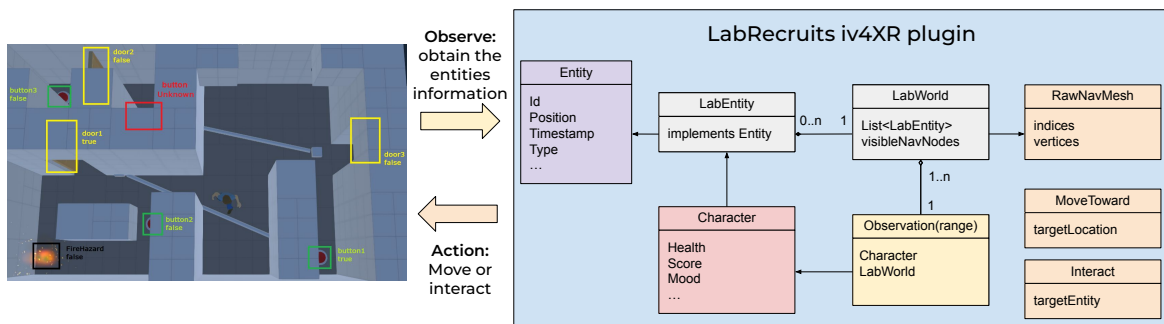


Fig. 2.7 LabRecruits-plugin overview

2.2.2 Observing the game environment state

In the IV4XR framework, a virtual game environment consists of a collection of *entities*. Each of these entities contains a set of properties that typically includes its unique game identifier, virtual location, and a list of additional name-value pairs that describe other attributes or state characteristics of the game. Depending on the inherent nature of the game, these entities can be independent game objects or game structures comprising sub-entities.

In the LabRecruits experimental game environment, each game object is represented as an entity that contains a **unique id**, resides in a **position**, contains a **timestamp** indicating the last instant the entity was sampled, belongs to a specific **type** (e.g., Character, Door, Button, Fire), and includes other properties such as whether the entity is **dynamic** or the **velocity** of moving entities. The *LabWorld* is the primary game structure, containing the list of entities that compose the LabRecruits game. The *Character* is a playable entity with additional properties such as **health**, **score**, and **mood**. Moreover, as explained in the next

⁴<https://github.com/iv4xr-project/iv4xrDemo>

navigation section (see subsection 2.2.4), the *LabWorld* also contains the visible navigable nodes on which the *Character* entity can move. Figure 2.8 represents how the *Character* and *LabWorld* game entities that constitute a game state have similarities with the widgets trees from GUI states.

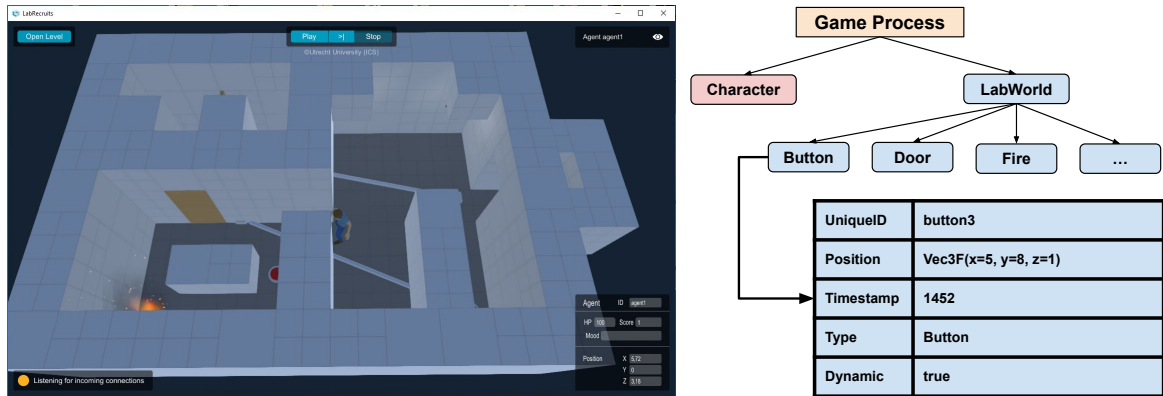


Fig. 2.8 The LabRecruits entities that constitute the game state

The TESTAR *agent* can connect with the *Character* to actively *Observe* all the game entities that reside in an observation range area. Thus, the game state depends on the observed entities and their properties. The *Character* is always present. There is always a *LabWorld*, but the existence of entities and visible navigable nodes depends on a configurable observation range of the *agent*, its distance from the game objects, and the blocking 3D walls.

Figure 2.9 shows how the observation range, a 3D sphere, works in the LabRecruits game environment. In this example, the *agent* observes itself, one button, and navigable nodes within the *LabWorld*. If the *agent* is near other entities or the height of the walls does not obstruct its observation, the *agent* would also be able to observe the existence and properties of other entities, such as a Door or Fire entities.

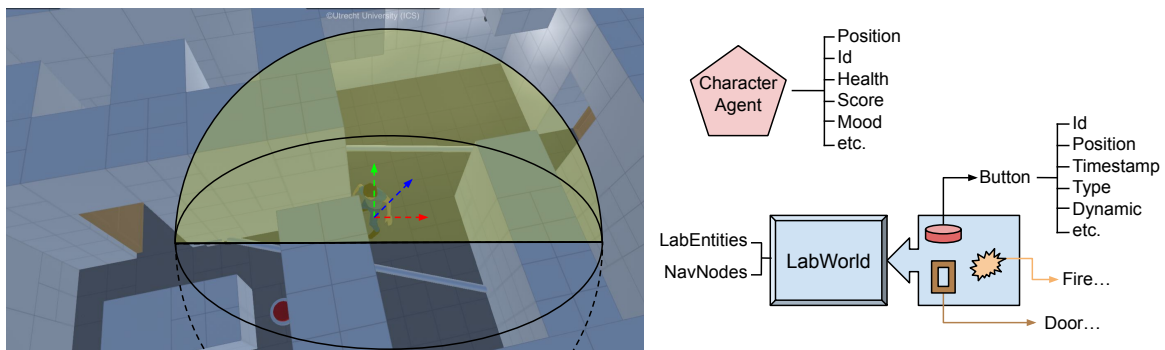


Fig. 2.9 Representation of the observed LabRecruits environment by the agent

In the IV4XR extension for TESTAR, the Widget, State, and Action classes are extended to IV4XRWidgetEntity, IV4XRState, and IV4XRAction (see Figure 2.10). These IV4XR taggable classes encompass IV4XR properties common to diverse game entities (i.e., position, identifier, name), which are defined in the final class IV4XRTags. However, since each game system may be inherently different, containing different game entities and performing diverse game actions, it is necessary to define specific Game-Taggable final classes to attach properties that are specific to a game-SUT through a game IV4XR-plugin.

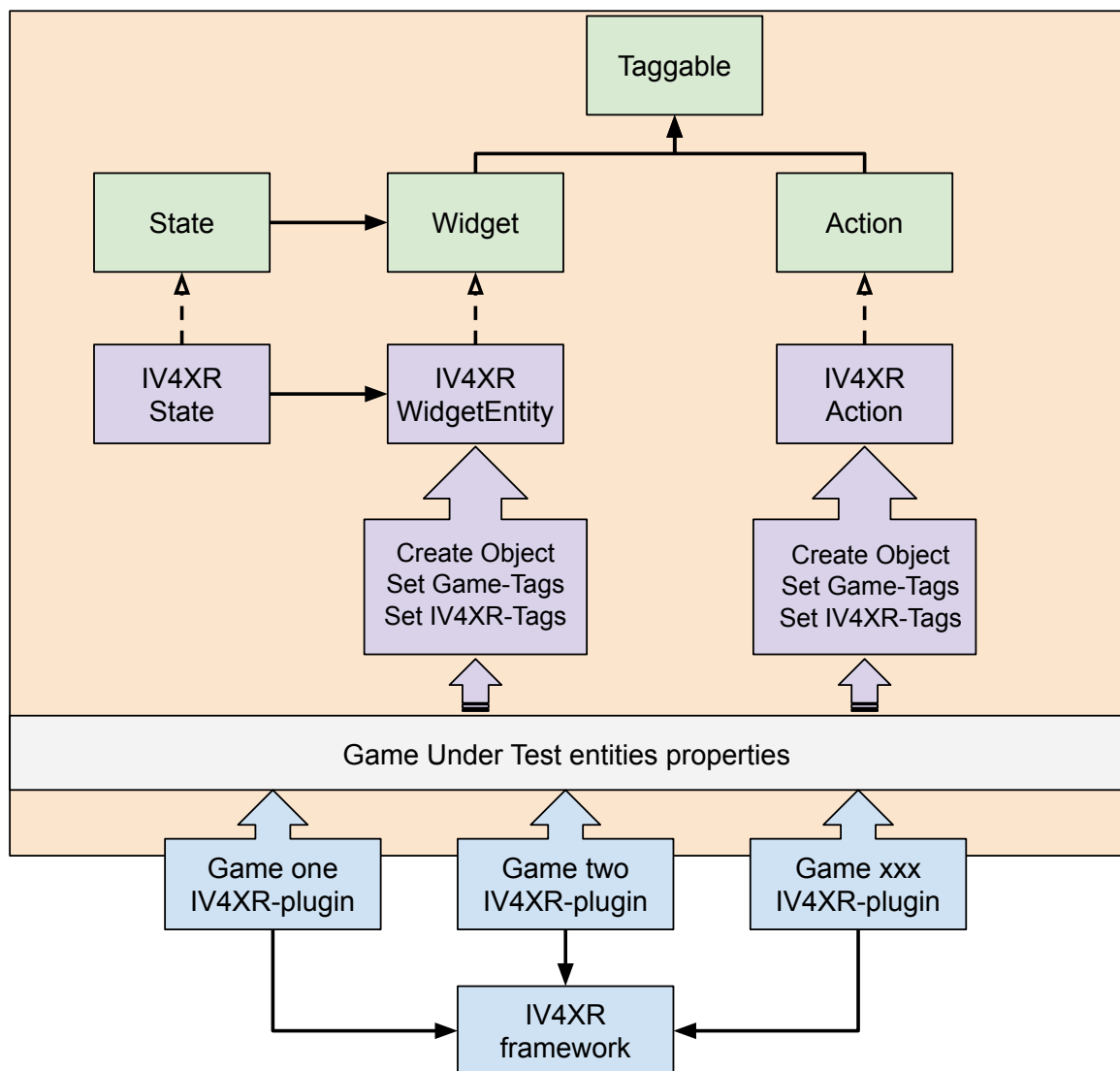


Fig. 2.10 Taggable IV4XRWidgetEntity, IV4XRState, and IV4XRAction classes obtained from a game-SUT plugin that implements the IV4XR framework interfaces

Extending the game state information

In a Windows system, TESTAR can use the Windows API native calls to identify the game-SUT process and monitor the CPU and memory usage of the game during testing. Moreover, it is also possible to use the functionality that enables the connection and extraction of data from local log files.

2.2.3 Deriving game environment actions

In virtual games running on desktop, web, or mobile systems, users can interact with the game through mouse and keyboard devices. Mouse coordinates and click events are used to select or interact with game entities, while the keyboard enables a wide range of game interactions via different key input events. Nevertheless, these traditional interaction methods, typically used for GUI test automation, present significant challenges when interacting with complex game environments.

In 3D game environments, characterized by inherent dynamics and non-deterministic behaviors, precise rotation, aiming, and movement controls are crucial for accurate navigation and interaction. Human users naturally perceive and adjust the orientation and position of their game characters within these environments. However, automation tools that rely solely on keyboard and mouse inputs fall short of meeting the necessary game interaction requirements if they cannot simulate this perception and control adjustment.

Game commands and tactical actions

The iv4XR framework addresses these challenges by enabling the observation of game internals to adjust an *agent's* position and orientation with a combination of movements, rotations, and interaction *commands*. Unlike conventional methods that rely on keyboard inputs, the iv4XR *commands* allows these adjustments by directly specifying the 3D position and orientation the *agent* needs to reach, facilitating more precise and effective interaction in test automation.

Similar to existing game entities and their properties, the available *commands* for interacting with a game may have similarities but are intrinsically different depending on the inherent nature of the game. The LabRecruits *commands* integrated into TESTAR include:

- Observe the game environments to obtain the information of the game state.
- MoveToward a target location specified as a 3D position.
- Interact with a near entity.

Observing the entity's data within game environments and executing *commands* allows the *agent* to move toward locations and interact with entities. However, *commands* may be insufficient for even simple game tasks, such as reaching a game entity in a 3D scenario to interact with it. To address this, it is necessary to group sequences of *commands* into *actions*. The iv4XR framework facilitates the creation of these *actions* using tactical programming [225]. These tactical *actions* enable test *agents* to combine multiple *commands* to, for example, follow a sequence of positional locations that lead to the target interaction entity.

2.2.4 Navigate game environments

Moving the *agent* within the game is a complex task, as it requires the *agent* to perceive which positions are obstructed/walkable to determine a path of positions to reach the desired entity. In the iv4XR framework, this functionality is known as *navigation*.

Some game engine platforms, such as Unity, can automatically build a *navigation* mesh⁵ that contains the walkable positions of the game by using the virtual objects geometry. Pathfinding algorithms can then optimize the traversal of these navigable mesh nodes to reach a desired position [70]. The iv4XR framework provides an A* pathfinding algorithm to efficiently find the best path between two nodes within a *navigation* mesh (NavMesh). The combination of tactical *actions* using the A* pathfinding algorithm over a *navigation* mesh allows the *agent* to effectively reach target locations and entities.

The LabRecruits game is built with Unity, enabling the retrieval of the raw *navigation* mesh of the loaded scenario. Figure 2.11 represents how the *agent* can observe a set of *navigable* positions and the path of actions that can be followed to reach each position.

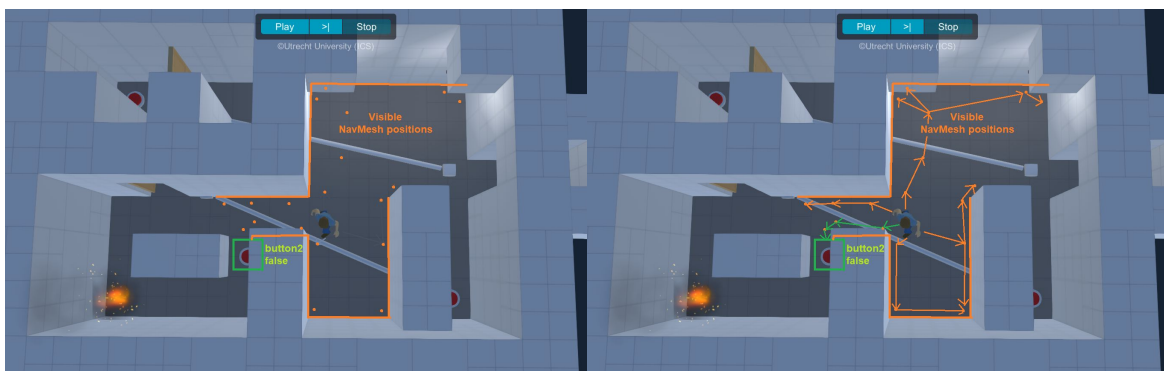


Fig. 2.11 LabRecruits available *navigation* mesh positions

⁵<https://docs.unity3d.com/es/Manual/Navigation.html>

2.2.5 Virtual environments for game software interaction

Since the TESTAR tool can be executed either as a local *agent* in a VM or packaged within a Docker container, its interaction with game systems largely depends on the specific requirements of the game-SUT and the available interaction *commands* for the various iv4xr plugins. For example, the LabRecruits game can be run with graphics disabled, enabling both LabRecruits and TESTAR to be containerized. In this setup, interaction *commands* can be sent to the game's listening process port within the container. On the other hand, Space Engineers requires running the game instance with its corresponding plugin in a VM. In this case, the TESTAR instance can be executed either within the same VM or on a remote VM or container, connecting to the Space Engineers plugin via IP address and port.

2.3 Summary

This chapter explains how the TESTAR tool interacts with a diverse range of desktop, web, and mobile GUI systems by using technical accessibility APIs and frameworks. It also details how TESTAR can be extended by integrating the iv4XR framework to interact with a 3D computer game.

Although the widgets or entities in GUIs and game states are technically different, they exhibit similarities. These differences stem from variations in the technical GUI plugins or the inherent distinctions of games, which involve different game objects.

In terms of action interactions, GUI actions for desktop and web systems primarily consist of native mouse and keyboard input events. Additionally, web and mobile systems can enhance interactions through script-based events. For game systems, internal game *commands* can be used instead of relying on traditional mouse and keyboard inputs. Moreover, tactical actions allow agents to combine multiple *commands* to navigate the complexities of 3D environments.

Chapter 3

TESTAR: Software exploration

When testing Software Under Test (SUT), an ideal exploratory strategy would aim to cover as many quality characteristics as possible in almost all SUT states or identify and report the states or actions that lead to failures. However, given the complexity of modern software systems and the vast number of potential states and actions, the reality is that we cannot predict with certainty where, when, how, or even if bugs will manifest. For this reason, a scriptless exploratory strategy should attempt to discover as many states and execute as many different actions as possible to maximize test coverage.

Once the scriptless tool has obtained the state of the application and derived the possible actions to execute, it is necessary to determine which actions to select depending on the desired exploration strategy. The Action Selection Mechanism (ASM) determines the sequence of instructions for exploring a SUT. An ineffective ASM can result in large portions of the application remaining uncovered and, consequently, untested.

The most basic ASM, which has proven effective in covering unexpected paths and finding undiscovered SUT failures, is the random ASM. After deriving the appropriate actions, applying a random algorithm in the TESTAR tool [54, 237, 5] and other state-of-the-art scriptless tools [60, 292] makes it possible to achieve considerable test coverage and find faults in industrial environments. However, the random ASM lacks exploratory efficiency because it requires a high amount of execution time to be effective in coverage [39, 14].

In previous TESTAR studies, a Q-Learning ASM, based on a Reinforcement Learning (RL) technique, was integrated to adjust the probability distribution of selecting non-executed actions by reducing the reward for actions that had been executed more frequently [34, 35]. Regarding coverage and fault detection, studies in the literature demonstrate that RL algorithms are generally more effective than random approaches [282, 137]. Nonetheless, when comparing TESTAR ASMs [80], or even comparing the TESTAR tool with other

state-of-the-art tools [219], evaluation results show that there are situations in which RL ASMs may have worse results than random ASM.

When compared to random strategies, the promising yet uncertain results of Q-Learning ASMs indicate that RL techniques continue to be a valuable area of research for enhancing TESTAR's exploration capabilities. For instance, recent studies explore the combination of reward functions [242] to address undesired behaviors such as jumping between states instead of exploring unvisited areas [206]. However, to further improve exploration efficiency and effectiveness, we believe it is necessary to study complementary exploration solutions, such as investigating an appropriate abstraction mechanism to identify stable states and actions, integrating exploration memory, implementing strategies that mimic human-like actions, and initiating research into extending exploratory ASM beyond GUI systems with eXtended Reality game environments.

This chapter aims to integrate exploratory strategies to enable the TESTAR tool to (dis)cover new states and actions more effectively and efficiently. While the existence and location of bugs in a SUT are unknown, applying exploratory strategies that automatically cover a large search space can help assess the SUT's robustness. Section 3.1 describes the mechanism for identifying the states and actions, the persistent state models that add memory support to the TESTAR tool, and the abstraction challenges when inferring a SUT model. Section 3.2 presents the integration of grammar-based Action Selection Rules (ASR) into TESTAR to prepare scriptless exploration that simulates a human-like behavior for web form-filling. Section 3.3 introduces an ASM that uses the observation, tactical actions, and navigational capabilities provided by the iv4XR framework to enhance spatial exploration effectiveness for 3D game environments. Finally, Section 3.4 summarizes the achieved results of this scriptless exploration chapter.

3.1 State and action identification for state model inference

In various studies within the GUI testing literature that explore the use of ASMs [168, 170, 282], it is discussed how the identification of widget properties influences state representation and the use of abstraction mechanisms to create abstract states by selecting only a subset of properties. TESTAR studies [34, 35] further emphasize the importance of assigning stable identifiers to each state and action to recognize them, even across different runs of the SUT. If an exploratory ASM is not applied over a stable foundation that adequately identifies the system's states and actions, the algorithm's effectiveness diminishes, becoming random or even counterproductive by repeatedly selecting the same action in a loop.

State-transition graphs and an initial application of abstraction mechanisms were integrated into TESTAR to provide visual information on test runs, enhancing the visualization and understanding of the testing process [246]. Additionally, these graphs also served as offline oracles, enabling TESTAR to apply queries that validated accessibility properties of the SUT from which the graph was inferred [73]. This state-transition graph implementation can be extended as persistent state models to add memory to the TESTAR tool, support exploratory ASMs, and research the definition of stable state and action identification and GUI inference challenges, such as dynamism and non-determinism.

3.1.1 Main mechanism for state and action identification

TESTAR uses the properties of the widgets (i.e., GUI widgets or XR widget-entities) to identify *concrete* and *abstract* states. **Concrete state identifiers** are calculated based on **all** the integrated properties (i.e., the properties obtained from the technical plugins) of **all** the widgets within the widget tree in a specific state. If any value of one of the widgets' properties changes, TESTAR identifies a new concrete state. This concrete state identifier allows precise detection and mapping of any changes in the state. However, for real and complex systems, the dynamic and highly modifiable properties of widgets make the concrete identifier a technique that generates extensive combinations of concrete states.

Some systems include functionalities that update and provide dynamic information over time without human or automated interaction. For instance, the left desktop GUI system in Figure 3.1 shows GUI widgets that dynamically update properties such as recording time or CPU consumption, delivering real-time information to users. Similarly, the right XR game system demonstrates dynamic astronaut properties that change over time due to simulated oxygen and energy consumption in space.

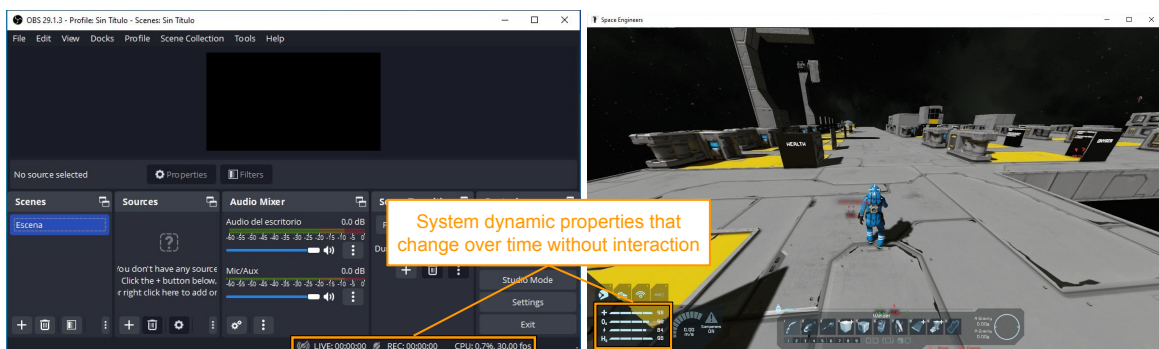


Fig. 3.1 Dynamic widgets properties of a desktop GUI and XR game systems that can constantly change without performing user or automated interaction

Other systems feature highly dynamic widgets that change with each user interaction. For example, the left desktop GUI system in Figure 3.2 shows GUI widgets that dynamically update after each user interaction, displaying selected volume in dB and sync offset in ms. On the right, the XR game system highlights how an astronaut's speed and position properties dynamically change whenever the astronaut moves within the virtual environment.

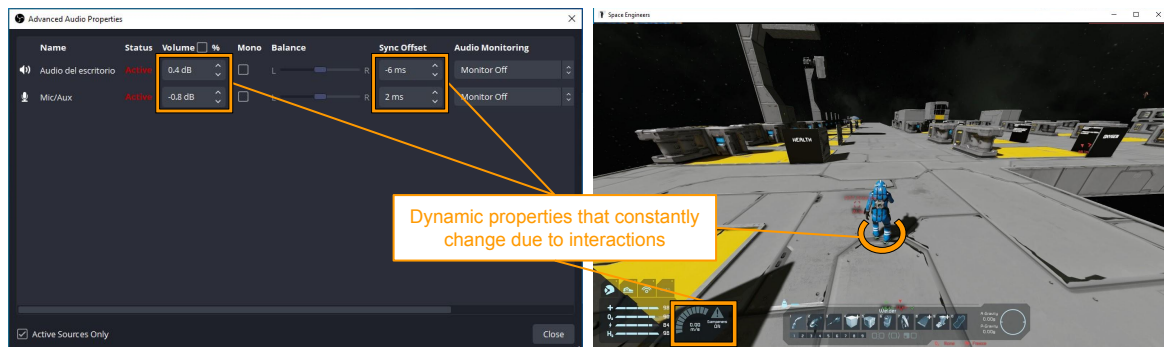


Fig. 3.2 Dynamic widgets properties of a desktop GUI and XR game systems that can constantly change after each automated or user interaction

Concrete action identifiers are calculated based on the concrete state in which the action was executed, the screen coordinates or 3D position of the interacted widget, and the role of the action (e.g., click, type for GUI systems or move, rotate, select item for XR game systems). If the action involves typing text, the specific characters typed are used to identify the action uniquely. For example, an action that types *foo* has a different concrete identifier than the action that types *boo*. Figure 3.3 illustrates how a single text field can lead to an infinite combination of text characters, depending on the maximum number of characters allowed. Consequently, the possible concrete action identifiers can also tend toward infinity.

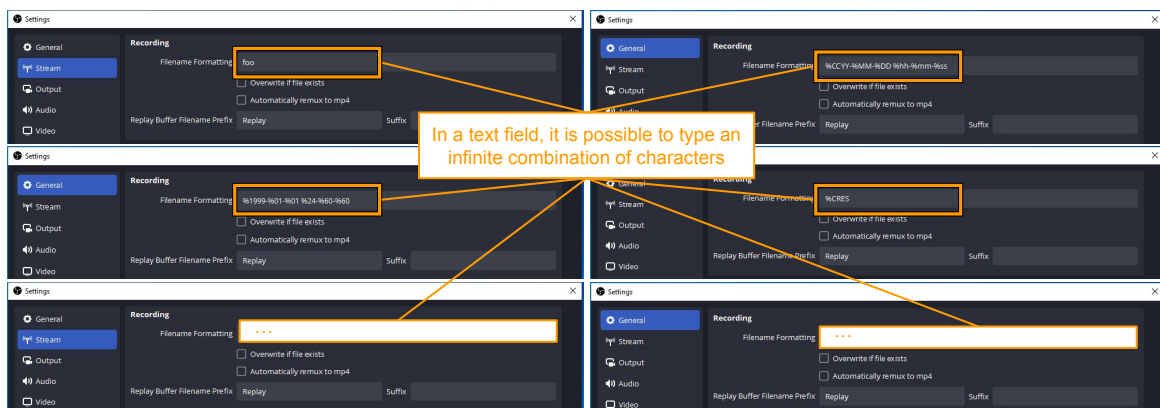


Fig. 3.3 A concrete action identifier based on typed text character tends to infinity depending on the maximum number of characters allowed

Attempting to concretely identify states with widgets that have dynamic properties would result in TESTAR continuously discovering new states. Additionally, differentiating writing actions based on the text would generate an unmanageable number of possible concrete actions for a simple text field. This continuous combinatorial identification of interactions and states, known as the state space explosion challenge, might drive the exploration to nearly infinite interaction paths or scenarios that can be performed during testing [195, 189]. To effectively manage state and action identification in dynamic, complex systems, TESTAR calculates an *abstract* identifier based on a selected subset of the widgets' properties.

Abstract state identifiers are calculated based on a selected subset of the widgets' properties. The subset can be configured by selecting which widget properties are used for state abstraction. The mapping between concrete and abstract states is surjective, meaning that each concrete state is mapped to one abstract state, whereas multiple concrete states can map to the same abstract state.

Abstract action identifiers are calculated based on the abstraction state in which the action was executed, the path of the interacted widget in the widget tree, and the role of the action without considering the text characters for type actions.

To enable the configuration of a subset of widget properties for state abstraction, TESTAR allows users to customize the *main abstraction mechanism* to select the most stable or suitable combination of widget properties.

Main abstraction mechanism

Since each SUT is designed and implemented differently, TESTAR allows users to select the widget properties they consider stable or suitable for identifying an abstract state. By following the previous example shown in Figure 3.1, the dynamic `Widget Title` property from the desktop GUI system on the left and the dynamic `Agent Oxygen` and `Agent Energy` properties from the XR game system on the right might be considered unsuitable for the main abstraction mechanism.

Similarly, in the Parabank web application from Figure 3.4, users may consider it inadequate to use the `Web Text Content` property for the main abstraction mechanism since it could provoke a state explosion due to the high combination of possibilities created by the amount of transferred money between account numbers after executing transfer funds actions. Otherwise, the role of the web element combined with the `Web Id` property (i.e., the span element with id "amountResult" and the span element with id "fromAccountIdResult") could be considered an appropriate main abstraction mechanism since these properties remain stable even with different combination of transfer funds actions.

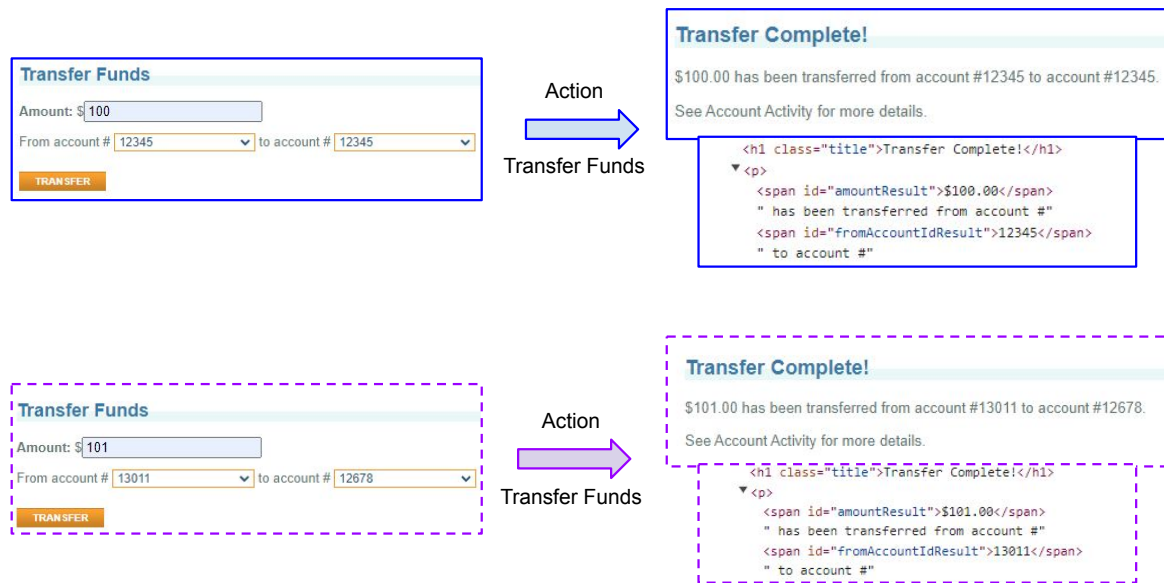


Fig. 3.4 Example of web widgets properties from Parabank that can be selected for the main abstraction mechanism

However, selecting a suitable abstraction mechanism by using a combination of widget properties is not trivial but a challenging task. This complexity arises because the abstract identification mechanism in large and complex systems is significantly affected by the challenges of dynamism and non-determinism.

Dynamism poses challenges in handling the dynamic values of widget properties. Including a dynamic property into the main abstraction mechanism can lead to increasingly detecting new states, resulting in a state space explosion. Moreover, an ASM that relies on abstract identifiers to determine the next action to select could perform randomly if it consistently misidentifies new states due to dynamism. On the other hand, deciding to exclude dynamic widget properties from the main abstraction mechanism may provoke non-determinism.

Non-determinism presents another challenge when configuring the main abstraction mechanism for state identification. A scenario is deemed deterministic when a specific action executed in a particular origin state consistently creates a transition to the same target state. However, excluding a key widget property that helps to differentiate two states in the main abstraction mechanism can provoke non-determinism.

For instance, Figure 3.5 exemplifies non-determinism in the OBS desktop application. The Widget Title property might be considered unsuitable and excluded from the main abstraction mechanism because OBS contains a widget that dynamically changes the title CPU consumption value. However, excluding this title property—used to compute the state

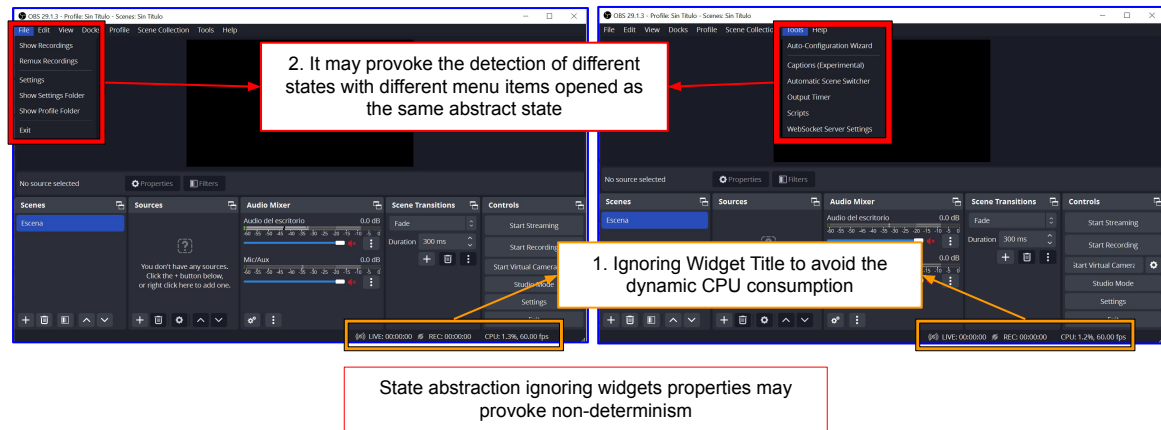


Fig. 3.5 OBS non-determinism when deciding to ignore the dynamic Widget Title property

abstraction identifier—can induce non-determinism when opening two different menu states that contain six different menu items. If the abstraction only considers properties such as the widget role, these distinct state menus, each comprising six widgets with the role of menu items, might be abstractly identified as identical states.

Selecting an appropriate abstraction mechanism requires inspecting, identifying, configuring, and analyzing the effects of various combinations of abstract properties. Moreover, different systems may require different abstraction mechanisms. In the following state model inference section, TESTAR is used to evaluate the exploration effectiveness impact when combining various abstract properties and investigate if it is possible to determine the most suitable abstraction mechanism.

3.1.2 State Model Inference

In Chapter 2, it was explained how the state of various types of systems can be obtained as a widget tree, along with the derived actions that simulate user interactions. In the previous subsection, it was described how TESTAR enables the configuration of a suitable abstraction mechanism using state-widget properties and action information to create concrete and abstract identifiers. This subsection presents the TESTAR state model inference approach, which creates a memory map to track discovered states, executed actions, and unvisited actions. Integrating memory for intelligent ASMs is crucial to enhance exploration effectiveness and efficiency.

The state model is dynamically inferred as TESTAR explores the SUT, observing states and deriving actions during its automated exploration. In previous TESTAR versions, state-transition graphs for memory were saved into a file and kept in memory during TESTAR execution [246]. In this thesis, the state model is inferred into a graph database, using

OrientDB by default [73]. These models enable capturing the flow of interaction events that represent the GUI's execution behavior [176].

The models are defined in terms of vertices for representing the states and edges for representing the actions. Both vertex and edge classes are a document in their own right, allowing for the required storage of the SUT data in the document's attributes (e.g., widget, state, and action properties). These models allow for using the graph theory, for example, to navigate the graph by selecting edge-actions until a desired vertex-state is reached.

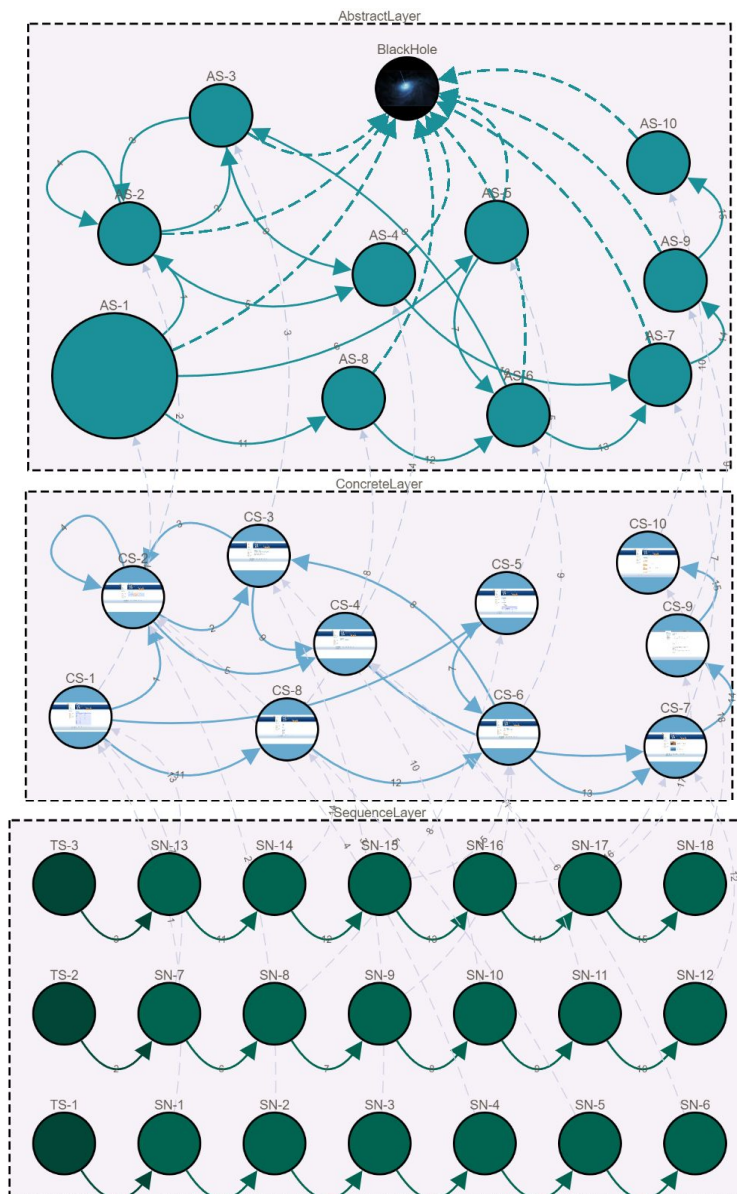


Fig. 3.6 Example of inferred state model from Parabank web application

TESTAR uses the state and action information to build the state model in three layers (see Figure 3.6):

- **Abstract layer:** It is the primary layer used to identify and create an abstract model for a specific SUT by using the subset of widget properties from the selected abstraction mechanism. It uses abstract state and action identifiers to create state-action transitions. The abstract model is built incrementally over multiple test runs. If the SUT name, SUT version, and selected widget properties in the main abstraction mechanism remain consistent, TESTAR connects to the existing abstract state model to continue the inference process. Otherwise, for instance, if a different abstraction mechanism is used, TESTAR creates a new abstract state model in the graph database.

The TESTAR tool uses this abstract model information to calculate navigation state-action transition paths and make intelligent action selection decisions. During the abstract model inference, when TESTAR arrives in a new abstract state and discovers abstract actions that have not been executed before, the tool uses a “BlackHole” state-vertex as a destination to mark unvisited actions.

- **Concrete layer:** It is also built incrementally based on multiple test runs and contains all the concrete state and action information extracted through the APIs used by TESTAR. Each state in the concrete layer is connected to a state in the abstract layer, and each concrete action is connected to an abstract action. The concrete model is linked to the abstract model. If TESTAR creates a new abstract state model, it automatically generates a new corresponding concrete state model.

In large and complex SUTs, the concrete state model will contain too many states and exhibit significant dynamism and non-determinism behaviors, making it impractical for driving TESTAR’s state-action transition navigation. Instead, this layer serves as a visual model for humans, aiding in the analysis of stored information when specific parts of the abstract model require deeper examination.

- **Sequence layer:** It is a management layer that records meta-information about the executed test sequences. Where the abstract and concrete layers describe the SUT behavior as state-action transitions, the sequence layer describes step-by-step the test sequences executed by TESTAR. The individual test execution steps (i.e., the action that transits to a state) are connected to the state nodes of the concrete layer.

In the abstract layer, TESTAR uses abstract identifiers of states and actions to create abstract model vertex and edges, respectively. If two states or actions produce the same abstract identifier, they represent the same abstract state-vertex or action-edge in the abstract

layer. In the concrete layer, TESTAR uses concrete identifiers of states and actions to create the corresponding concrete model state-vertices and action-edges. The TESTAR state model will likely contain many more unique concrete elements than abstract elements because the concrete identifiers incorporate many more widget, state, and action properties by default.

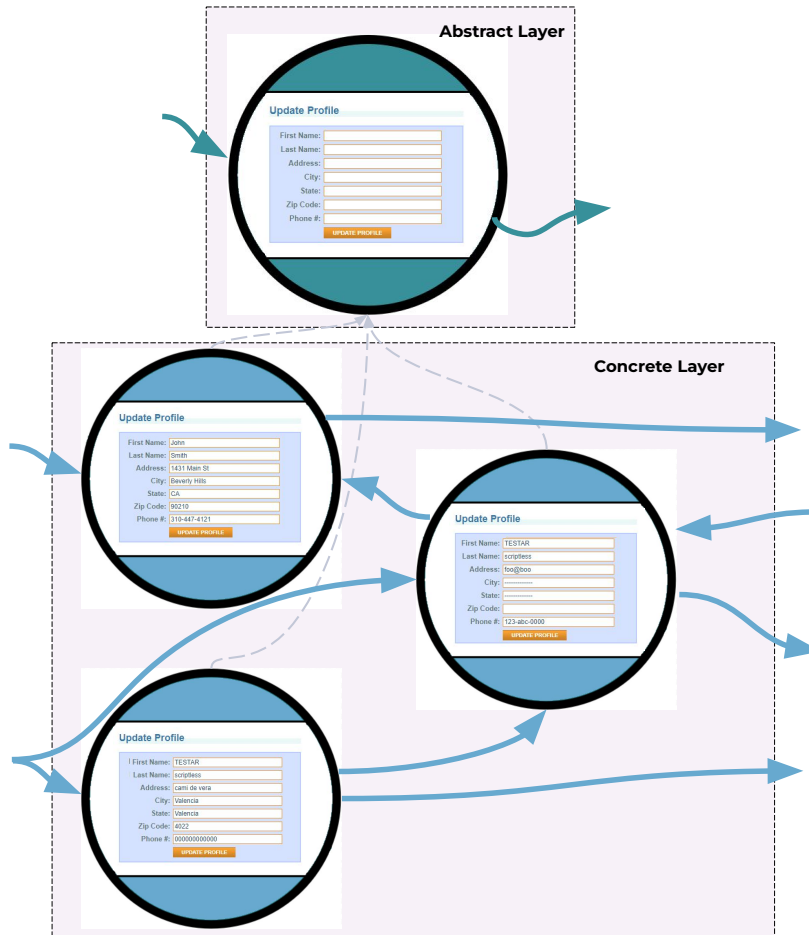


Fig. 3.7 Example of surjective abstract and concrete states mapping from Parabank web application

The mapping between concrete and abstract states is surjective, meaning that each concrete state is mapped to one abstract state, whereas multiple concrete states can map to the same abstract state. For instance, Figure 3.7 represents how an abstraction strategy considers the "Update Profile" abstract state the same regardless of the text in the form because the Web Text Content property is not used in the main abstraction mechanism.

Unvisited actions first

The Unvisited actions first ASM [189] observes the current abstract state for actions that have not yet been visited. For instance, as shown in Figure 3.8, consider the TESTAR exploratory agent is currently on the state AS-4 with a depth of 0. In this case, if AS-4 contains multiple unvisited actions, it selects one randomly. However, suppose there are no unvisited actions in the current state (e.g., AS-4 has no unvisited action to the black hole). In that case, TESTAR uses the state model information to find a path of actions that leads to the nearest state with unvisited actions. For example, the states near AS-4 that contain unvisited actions are AS-3 and AS-5, both with depths of 1. TESTAR will randomly select one of the unvisited actions from AS-3 or AS-5 and follow the path of actions until it reaches and executes the selected unvisited action.

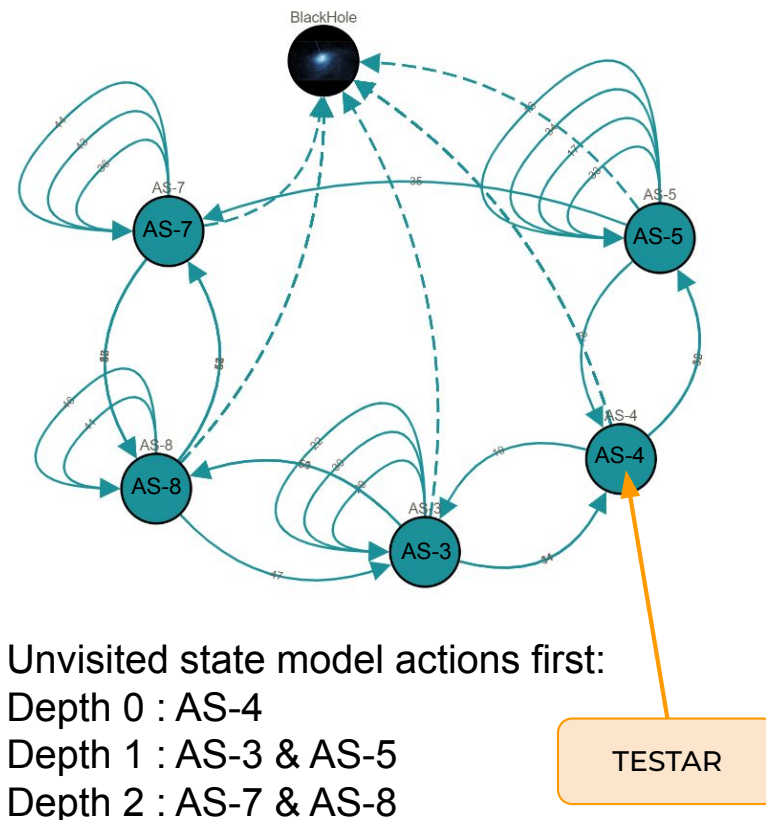


Fig. 3.8 Unvisited actions first ASM using the state model information

Algorithm **State_Model_ASM** was implemented in TESTAR to prioritize the selection of nearby abstract actions that have not yet been visited. The goal is to select a new action when TESTAR is in state s . It uses the *State_Model* that is being inferred to identify and maintain a *path* of actions that leads to a specific unvisited action it wants to prioritize. If the

Algorithm State_Model_ASM

```

Require:  $s$                                 ▷ The state the SUT is currently in
Require:  $State\_Model$                        ▷ The state model that is being inferred
Require:  $path$                                ▷ Path towards an unvisited action
1: if  $path \neq empty$  then                    ▷ ASM is following a previously determined path
2:    $a \leftarrow path.pop()$                    ▷ Selected action is next one on the path
3: else                                         ▷ If the path is empty, we will create a new one to an unvisited action
4:    $reachableStates = getReachableStatesWithBFS(s, State\_Model)$ 
5:    $unvisitedActions \leftarrow empty$ 
6:   while ( $unvisitedActions == empty \wedge reachableStates \neq empty$ ) do
7:      $s' = reachableStates.pop()$ 
8:      $unvisitedActions \leftarrow getActions(State\_Model, s', \mathbf{unvisited})$ 
9:   end while
10:  if  $unvisitedActions \neq empty$  then       ▷ Unvisited actions found with BFS
11:     $ua \leftarrow selectRandom(unvisitedActions)$    ▷ Randomly select an unvisited
12:     $path \leftarrow pathToAction(ua)$            ▷ Calculate path from  $s$  to walk to  $ua$ 
13:     $a \leftarrow path.pop()$                  ▷ Selected action in  $s$  is next one on the path to  $ua$ 
14:  else                                       ▷ No unvisited action found with BFS
15:     $availableActions \leftarrow getActions(State\_Model, s, \mathbf{all})$    ▷ Get all actions in  $s$ 
16:     $a \leftarrow selectRandom(availableActions)$ 
17:  end if
18: end if
19: return  $a$ 

```

$path$ is not empty (line 1) because the nearby unvisited action has been previously identified, the ASM maintains the $path$ and just selects the next action to execute. If the $path$ is empty because TESTAR has initialized the algorithm or because the previous path was already completed, the ASM tries to identify a new unvisited action. This identification is done by searching (in Breadth-first search (BFS) order) for $unvisitedActions$ (line 8) from all the states that are reachable from s in the state model (line 4). If unvisited actions are found, TESTAR randomly selects one ua (line 11) and updates the $path$ to reach the state where that action can be found for execution (line 12). Then, it selects the first action from the $path$ that leads towards that action (line 13). If no unvisited actions are found, the ASM just randomly selects an action from those available in state s (line 16).

Evaluation of the Unvisited actions first ASMs

This evaluation aimed to measure the exploratory effectiveness of the **State_Model_ASM** and compare combinations of abstract properties used in the main abstraction mechanism. The following research question was used to guide this evaluation:

- StateModel-RQ1: How do different levels of abstraction influence the exploratory effectiveness of State_Model_ASM in automated GUI exploration compared to Random_ASM?

Table 3.1 Java Access Bridge properties and the possible impact of using the attribute for state abstraction in Rachota

Attribute	API	Abstract representation impact
Title	Name	Visual name of the widget. In Rachota this is a dynamic attribute because widgets update the current time.
HelpText	Description	Tooltip help text of the widget. In Rachota this attribute is static.
ControlType	Role	Role of the widget. Cannot always distinguish different elements, and hence can cause non-determinism.
IsEnabled	States	Checks if the widget is enabled or disabled.
Boundary	Rect	Pixel coordinates of the widgets' position. Even one pixel change would result in a new state, so it was considered too concrete for the experiments.
Path	childrenCount + parentIndex	Position in the widget-tree. Useful for differentiating states based on the structure of the widget tree.

The Random_ASM was compared with 4 levels of abstractions using the State_Model_ASM. Table 3.1 shows the widget's properties from the Java Access Bridge API implemented in TESTAR for Java Swing applications. Based on these properties, the following levels of abstraction were established:

1. Abstract: ControlType
2. Intermediate: ControlType, Path
3. Dynamic: ControlType, Path, Title (including the dynamic attribute Title)
4. Customised Abstraction: ControlType, Path, HelpText, IsEnabled (this one was customized for Rachota following the impacts described in Table 3.1)

Dependent variables

To answer the research question, Random_ASM and State_Model_ASM effectiveness were measured in terms of GUI coverage and code coverage. GUI coverage indicates the number

of discovered model states. Code coverage (instruction and branch) is measured using JaCoCo [127]. Both metrics were collected after each executed action.

SUT Object

To be able to measure code coverage, the open-source Rachota [198] application was used as the SUT. It is a Java Swing application for time tracking different projects (see Figure 3.9). The details of Rachota are presented in Table 3.2.

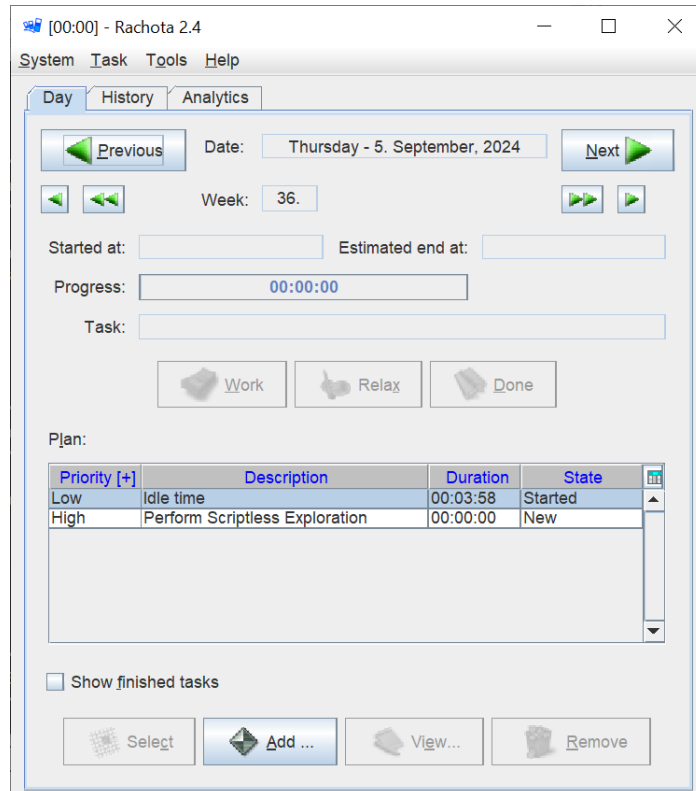


Fig. 3.9 Rachota Timetracker SUT

Table 3.2 Details of Rachota SUT

Java Classes	52
Methods	934
LLOC	2722
Inner classes	327

Design of the experiment

Each test run contains one sequence of 300 actions, which is considered enough to show the differences between TESTAR ASMs [274]. To be able to form valid conclusions and to deal with the randomness, each test run was repeated 30 times [18].

Results

The results of the code coverage measurements are in Figure 3.10. They show that the level of abstraction affects the GUI exploration effectiveness of the State_Model_ASM. Having too high or too low level of abstraction negatively impacts the exploratory effectiveness.

Overall, the average coverage of all State_Model_ASM abstraction mechanisms outperformed the Random_ASM, even with a less suitable abstraction. This means that model-based ASMs are a promising way to improve the effectiveness of scriptless testing. However, it is important to note that the best random sequences can achieve similar coverage to the average coverage provided by abstract, dynamic, and intermediate mechanisms. This underscores the relevancy of customizing a suitable abstraction mechanism to enhance ASM effectiveness compared to random and highlights the inherent capabilities of random exploration itself.

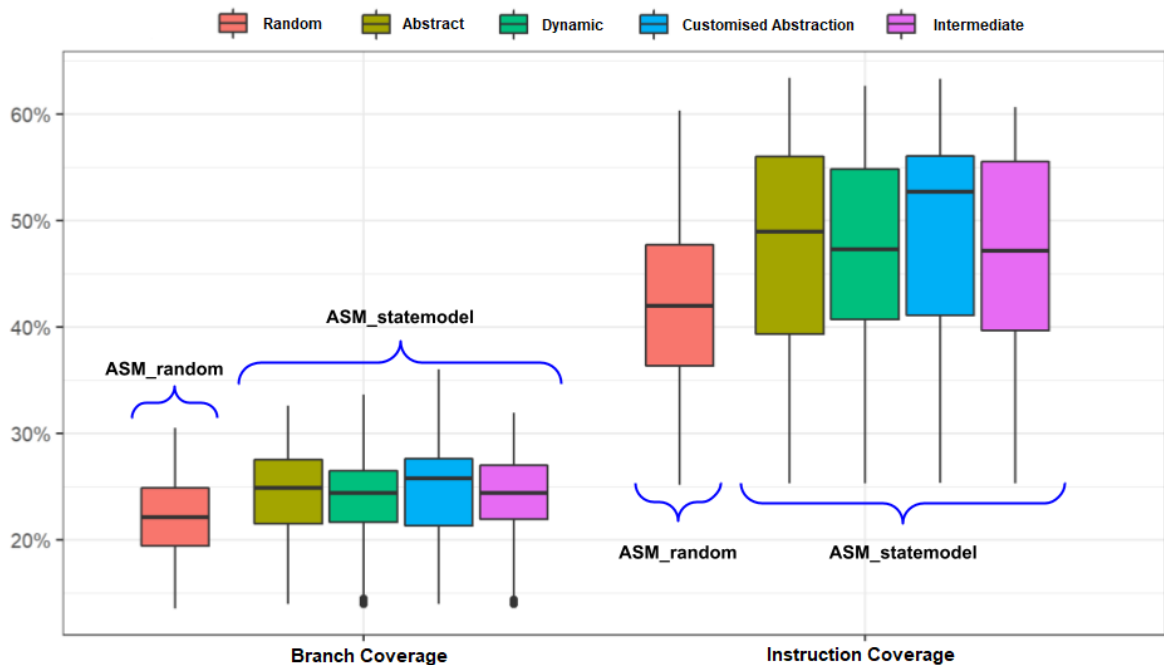


Fig. 3.10 The code coverage (%) that was reached when comparing the Random_ASM with 4 different abstraction levels of the State_Model_ASM

To investigate the impact of the changing levels of abstraction used in the experiments on the number of abstract and concrete states created, longer test runs of 3000 actions were performed. Again each run is repeated 30 times for each configuration.

The results are shown as a box plot in Figure 3.11. For the concrete states, the customized level that identifies the most concrete states demonstrates slightly better GUI exploration capabilities, aligning with the code coverage results. In terms of abstract states, the customized level creates more abstract states compared to abstract and intermediate configurations but

significantly less than the dynamic one. However, the dynamic mechanism, while identifying a greater number of abstract states, does not necessarily improve the exploratory effectiveness. This is because the dynamic configuration, which includes the dynamic widget title as part of the abstraction mechanism, considers that new abstract states are constantly being discovered. This dynamism hinders exploration effectiveness, as the abstract mechanism fails to adequately track which states are newly discovered versus those that already exist during scriptless test exploration.

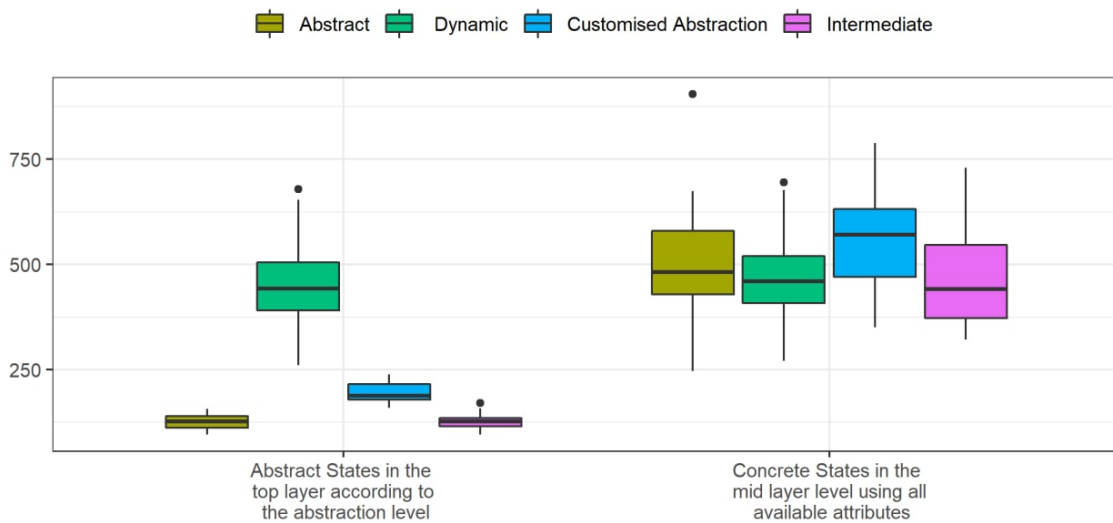


Fig. 3.11 The number of abstract states (top layer) and concrete states (mid layer)

Therefore, based on the obtained results, we can answer the `StateModel-RQ1`: *How do different levels of abstraction influence the exploratory effectiveness of `State_Model_ASM` in automated GUI exploration compared to `Random_ASM`?* We conclude that using different levels of abstraction significantly influences the exploratory effectiveness of the GUI exploration process. Moreover, configuring a suitable level of abstraction improves the effectiveness of GUI exploration, as measured in code coverage.

Experience configuring a suitable level of abstraction

A main challenge when configuring the main abstraction mechanism of TESTAR is to select a suitable subset of widget properties for state abstraction that deals with dynamism and does not cause state explosion. However, various studies involving experiments with Rachota and additional evaluations with the Notepad desktop application [188, 189] reveal that dynamism and state explosion are not only the result of dynamic values but can also arise from how widgets change due to SUT-specific characteristics:

- **Dynamic increment of widgets:** In some applications, such as Rachota, dynamic lists of elements can continuously add new task items. This creates new widgets and states in the model, leading to state and action explosions.
- **High number of combinatorial elements:** In certain applications, like Notepad, multiple scroll lists with a large number of different elements generate combinatorial displays of widgets. While it may not be functionally important to cover all these options, it creates multiple states in the model (e.g., the Font selection in Notepad).
- **Slide actions:** In some applications where scrolling actions are required, the exact scrolling coordinates from start to end can cause a change in the number of widgets visible in the state. Depending on the state abstraction and how the widget tree is obtained, this can create new states and cause a high number of combinatorial possibilities.
- **Popup information:** In some applications, for example, Rachota, a descriptive popup message may appear for a few seconds in the GUI when the mouse is hovered over some of the widgets. This could result in a new state for the model, and it might cause non-determinism if hovering over a widget was not an intentional action that was executed on purpose.

Furthermore, attempts to use fewer or more properties for abstraction, or different combinations of them, consistently result in the creation of non-deterministic models. Since the *main abstraction mechanism*, which relies on a selected subset of the widgets' properties, proves inadequate for model inference when exploring complex systems, the following section outlines the use of *abstraction sub-strategies* to define a more effective SUT **abstraction strategy**.

3.1.3 Abstraction strategy for state and action identification

Abstraction is a complex and challenging aspect of testing approaches that use model inference techniques. Attempting to implement a stable abstraction strategy using only a main abstraction mechanism is not a viable solution for large, complex systems exhibiting dynamic and non-deterministic behaviors. Within the same SUT, widget properties are needed to distinguish between abstract states that may lead to non-determinism; however, these same properties may have dynamic values in other widgets, which can result in the identification of new states. For instance, as shown on the left side of Figure 3.12, excluding the widget `Title` property in Rachota may be a suitable decision to avoid dynamism. Nonetheless, as

depicted on the right side, the widget Title property is crucial to avoid non-determinism when distinguishing Rachota menu items.

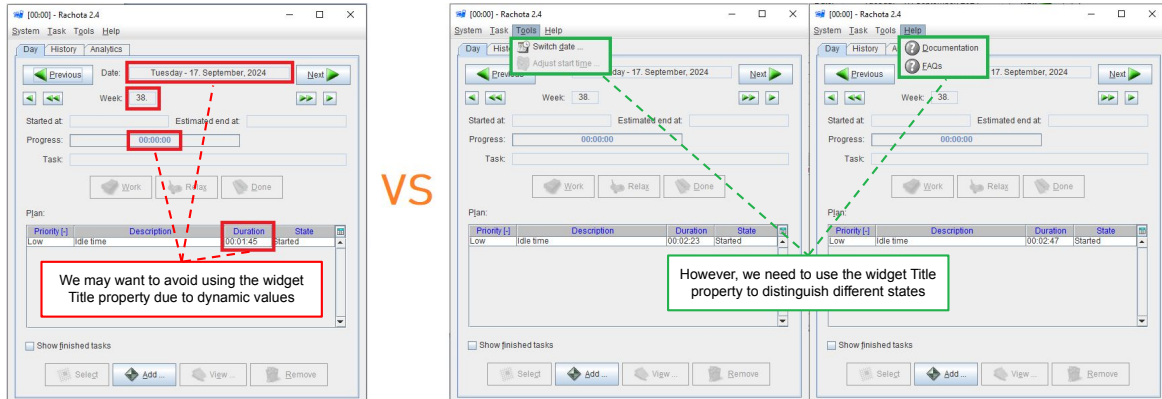


Fig. 3.12 The decision challenge of excluding or including the widget Title property in Rachota

This challenge is not limited to specific desktop applications like Rachota but is an inherent issue in large and complex systems. For instance, in the Parabank web application, the Web Id property appears to be a suitable candidate for inclusion in the main abstraction mechanism. However, as illustrated in Figure 3.13, there is a form phone widget that generates dynamic values for the Web Id property each time the state is visited. This can lead to state explosion, as Bill Payment Service states may be incorrectly identified as new each time they are visited.



Fig. 3.13 Web widget in Parabank that uses a dynamic Web Id property

Moreover, this challenge becomes exacerbated by the presence of fully dynamic widgets within a SUT. Complex systems allow adding, modifying, and removing data as dynamic widgets with properties. For example, Figures 3.14 from Rachota and 3.15 from Parabank illustrate dynamic row widgets within tables. In Rachota, each time a task is created, the system dynamically adds new row widgets to the table. Similarly, in Parabank, whenever a

new account is created, a transaction is initiated, or a bill payment is processed, the system dynamically adds new row widgets to the table.

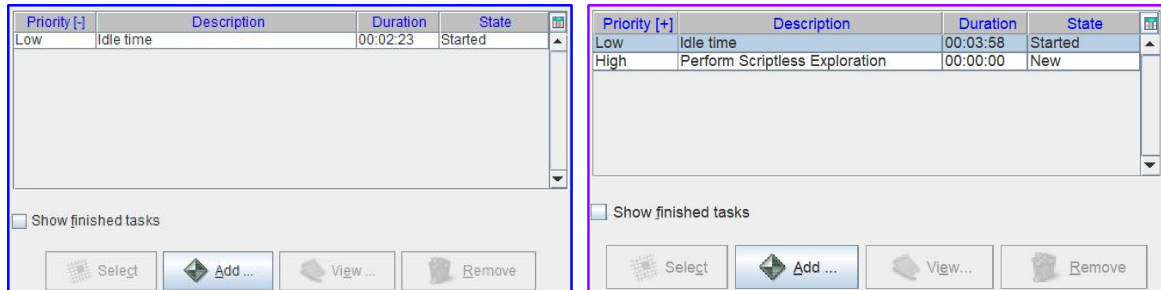


Fig. 3.14 Desktop widget (table) in Rachota that contains dynamic widgets (table rows)

Date	Transaction	Debit (-)	Credit (+)
05-06-2024	Funds Transfer Sent	\$100.00	
05-13-2024	Funds Transfer Sent	\$100.00	
06-04-2024	Funds Transfer Received		\$101.00
06-04-2024	Funds Transfer Sent	\$101.00	

Fig. 3.15 Web widget (table) in Parabank that contains dynamic widgets (table rows)

These efforts to identify dynamic property values and widgets aim to infer deterministic models while dealing with state explosion. However, non-deterministic behavior may be an inherent challenge in a SUT due to a lack of information in the GUI. For instance, Figure 3.16 illustrates non-determinism behavior in the Parabank web application. The first time the action `Apply` for a `Loan` is executed with a specific amount of money for a specific account, there is a transition to a state indicating the bank has approved the loan. However, in a subsequent execution of the same action, the specific account does not have sufficient funds for the given down payment. This is considered the same origin state because there is a lack of information in the state to distinguish it based on the amount of available money in that specific account. Therefore, an action selection algorithm that relies on abstract identifiers combined with previously obtained memory may be negatively affected if the intention is to navigate back to the same particular state.

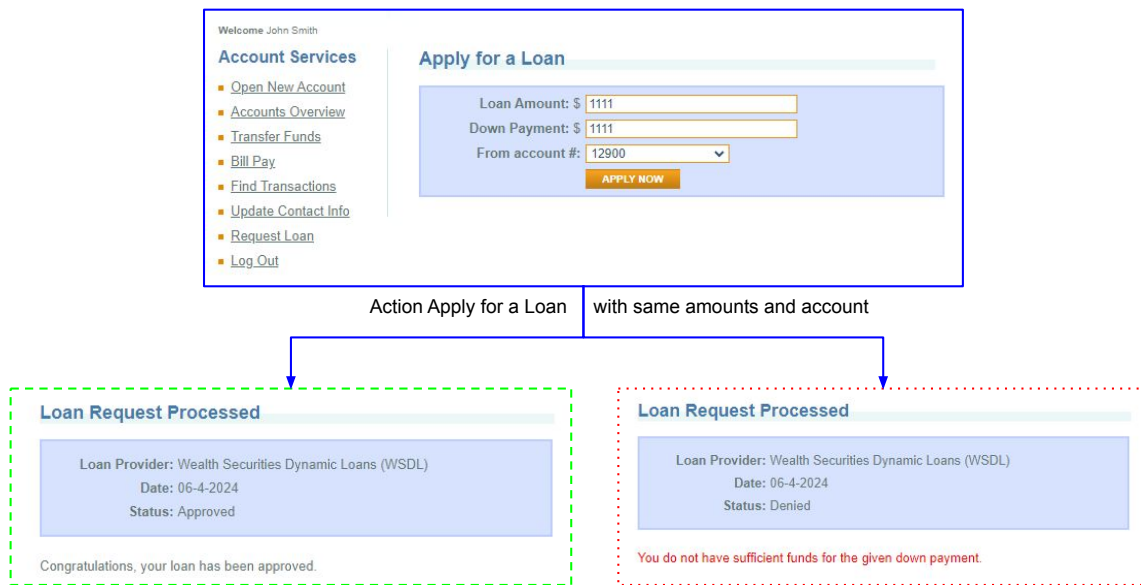


Fig. 3.16 Parabank non-deterministic transition after executing the same action in the same particular state

Abstraction strategy including sub-strategies

The nature of large and complex systems makes the main abstraction mechanism and the selection of a subset of widget properties not sufficient to define a suitable abstraction strategy. It is necessary to define the TESTAR abstraction strategy as a combination of the main abstraction mechanism with the introduction of a set of *abstraction sub-strategies* to deal with the challenges that affect the state abstraction identification.

TESTAR allows to programmatically prepare a set of abstraction sub-strategies to deal with dynamic and non-deterministic challenges. For example, the `Web Id` property from the Parabank application can be used as the main abstraction mechanism, while an abstraction sub-strategy can be specifically prepared to ignore the dynamic values of the randomly generated format of the form phone widget, as shown in previous Figure 3.13. Similarly, another abstraction sub-strategy for testing Parabank can be prepared to ignore table row widgets, as illustrated in previous Figure 3.15.

As an example of abstraction strategy for Parabank, Figure 3.17 shows how the `Web Id` and `Web Name` properties are configured in the main abstraction mechanism, as they are considered the most stable widget properties. Additionally, a set of abstraction sub-strategies are programmatically applied to ignore dynamic row widgets and, customize specific properties for dynamic phones and select widget properties. This abstraction strategy represents a suitable combination for distributed test exploration in Parabank (see Chapter 4).

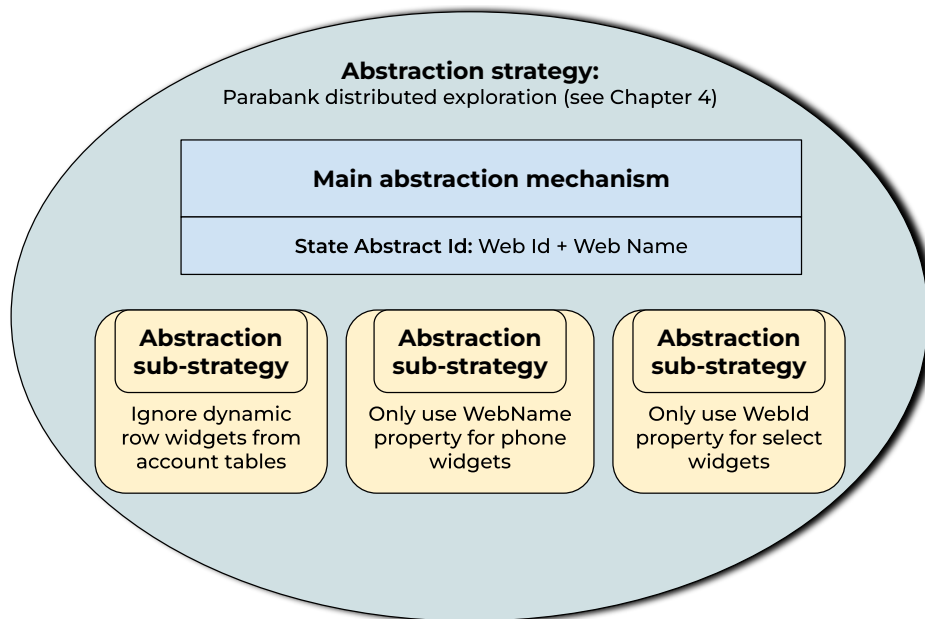


Fig. 3.17 Abstraction strategy used by TESTAR which combines a main abstraction mechanism with abstraction sub-strategies

Defining a suitable **abstraction strategy** requires users to inspect, identify, configure, and analyze the effects of combining the *main abstraction mechanism* with various *abstraction sub-strategies*. The considered suitable abstraction strategy may vary depending on the specific testing goals. For instance, an abstraction strategy for maximizing test coverage may differ from another focused on inferring deterministic models for applying offline oracle approaches. Additionally, different users may find that different abstraction strategies better suit their needs. As a result, there is no universally perfect abstraction strategy in TESTAR that applies to diverse systems. Instead, TESTAR offers a configurable approach that allows users to adapt abstraction strategies to their specific SUT and testing needs.

3.2 Grammar Action Selection Rules for Scriptless Testing

The TESTAR tool enhances scriptless testing by integrating memory frameworks and intelligent ASMs. Integrating a state model to prioritize unvisited abstract actions [189] or using Reinforcement Learning (RL) ASMs [80, 274, 242] significantly improves test exploration effectiveness in terms of state and code coverage when a suitable abstraction strategy is configured. Nevertheless, despite these advancements in automated test exploration, some features of today's applications are designed for humans to follow a series of continuous, specific steps to execute particular functions. For example, performing a bank operation may

require users to perform a specific sequence of steps by filling out one or multiple forms with particular values.

While intelligent algorithms, such as unvisited or RL ASMs, may improve the efficiency and effectiveness of exploration, they fall short in covering human-like strategies for complex tasks. Scriptless testing struggles with handling multiple human-like steps. The more numerous and specific the steps required, the greater the difficulty for the semi-random nature of an ASM to follow the correct sequence. For instance, a scriptless approach might start filling out form inputs but attempt to submit the form before all inputs are completed, or it might click a navigation menu that moves to a different state before completing the form.

Droidmate [41] and Humanoid [156] are scriptless tools for Android systems that use a model to predict the probability of a user choosing each possible action in the current state, and then these probabilities are used as a bias for random selection. The Behavior based Automated Testing approach (eBAT) [194] is an action selection strategy that defines a policy derived from the tester's interaction patterns with a specific web application. This policy is created by employing state abstraction, interdependent action grouping, redundancy reduction, and tree-based traversal of the system.

Grammar-based testing is a well-established technique that uses grammars to define and generate *test input data*. Several tools and methods use grammars for testing to automate the process of test input data generation [191, 263, 17, 20, 104]. However, grammars are not yet being widely used to guide testing strategies. This technique can enable a more human-like testing process and provides an effective way to test application components that require specific sequences of steps. Consequently, this novel application of grammars is an important contribution to TESTAR and to the field of scriptless testing.

In this section, we have integrated a scriptless testing action selection approach based on a context-free grammar that enables the specification of Action Selection Rules (ASRs) [122]. These ASRs allow testers to define human-like strategies within TESTAR, prioritizing actions based on context. This grammar aims to effectively translate user behavior or roles into rules that dictate the next action to be taken during execution, bridging the gap between automated exploration and the structured, sequential steps often required by modern applications.

3.2.1 Grammar-Based Action Selection Rules in TESTAR

The ASRs define how to select the next action from the possible actions in the current state of the SUT. The grammar for defining ASRs, described in the Extended Backus-Naur Form (EBNF) in Listing 3.1, includes structures such as weighted action instructions and if-then-else branches. These branches evaluate various aspects of the SUT, such as the type

of the SUT, if the state has changed, the relation of widgets in the widget tree, the type of available action, how many actions have been (un)visited, or some combination thereof.

```

<strategy> ::= <action> {<action>} | <if_else>
<if_else> ::= if <bool_expr> then <action> {<action>} else <action> {<action>}
<bool_expr> ::= not <bool_expr> | <bool_expr> <bool_opr> <bool_expr>
                | <int_expr> <int_opr> <int_expr> | <state_bool> | <bool>
<bool_opr> ::= and | or | xor | is
<int_opr>  ::= > | >= | < | <= | == | !=
<int_expr> ::= <n_actions> | <int>
<state_bool> ::= state-changed | any-exist [<mod>] [<filter> <relation>]
                | any-exist [<mod>] [<filter> <a_type>] | sut <filter> <sut_type>
<n_actions> ::= n-actions [<mod> <a_type>]
<action>    ::= [<int>] select-previous | [<int>] select-random [<mod>] [<filter> <a_type>]
                | [<int>] select-random [<mod>] <filter> <relation>
<filter>    ::= of-type | not-of-type
<a_type>    ::= click-action | type-action | hit-key-action | drag-action
                | form-field-action | form-submit-action
<mod>      ::= most-visited | least-visited | visited | unvisited
<relation> ::= sibling | child | sibling-or-child
<sut>      ::= windows | linux | android | web
<bool>     ::= true | false
<int>      ::= 0...9 {0...9}

```

Listing 3.1 The grammar to define strategies for action selection rules in EBNF

Once the grammar is defined, each semantic expression must be associated with the different types of widgets, actions, or other TESTAR logic. An implemented grammar map ensures that each semantic expression is associated with the relevant TESTAR logic, enabling context-aware decision-making. For example, Figure 3.18 presents an overview of how the grammar expression corresponding to action types (<a_type>) are mapped to the TESTAR internals that check if the action roles are from a specific type or if more complex action types like form-filling actions are from a specific type whereas the actionable widget is children of a form widget.

3.2.2 Grammar-based ASR in TESTAR for form-filling

To demonstrate the effectiveness of the TESTAR grammar-based approach, the proposed grammar was used to tackle form-filling in scriptless testing.

Grammar for (a_type)		TESTAR implementation for Types of Actions
click-action	↔	<code>action.get(Role) == ActionRoles.Click</code>
type-action	↔	<code>action.get(Role) == ActionRoles.Type</code>
hit-key-action	↔	<code>action.get(Role) == ActionRoles.HitKey</code>
drag-action	↔	<code>action.get(Role) == ActionRoles.Drag</code>
form-field-action	↔	<code>action.get(OriginWidget).get(Role).equals(WebINPUT) && action.get(OriginWidget).parent().get(Role).equals(WebFORM)</code>
form-submit-action	↔	<code>action.get(OriginWidget).get(Role).equals(WebSubmit) && action.get(OriginWidget).parent().get(Role).equals(WebFORM)</code>

Fig. 3.18 Implemented map that associates grammar action expressions with TESTAR actions

Web forms are ubiquitous components of modern web applications, serving various purposes such as login screens, questionnaires, and filtering options in e-commerce websites. However, as previously explained, scriptless testing struggles to handle web forms, especially those with numerous fields, as it is sometimes necessary to address all fields and inputs before submitting the form. Moreover, a single action executed outside the expected filling-form workflow can potentially undo all progress, making the testing process unreliable.

Listing 3.2 shows a grammar-based ASR that captures a human-like strategy to mimic form-filling. This ASR first prioritizes filling the visible unvisited fields, then scrolling down if possible to make more form widgets visible, and only once both have been completed it clicks the submit button. Note that there is no predefined order for selecting the unvisited fields to fill.

```

IF any-exist unvisited of-type form-field-action
THEN select-random unvisited of-type form-field-action
ELSE IF any-exist of-type scroll-action
  THEN select-random of-type scroll-action
  ELSE select-random of-type form-submit-action

```

Listing 3.2 A human-like grammar-based form-filling ASR

To integrate a grammar-based ASR, TESTAR needs to invoke the desired ASR instead of the default action selection decision-making. For example, TESTAR can explore a web SUT randomly or by following other ASM until a form is detected. Then, it will invoke the form-filling detection ASR defined in Listing 3.2 to select which action to execute.

Algorithm **Form_filling_ASM** was implemented in TESTAR together with the grammar-based form-filling ASR to perform a human-like strategy to fill forms with correct values.

For each state s of the SUT, if s does not contain any *form* widget (line 1), TESTAR derives *allActions* that can be performed for all existing widgets in s (line 2). Then, it randomly selects one of *allActions* to explore the SUT (line 3).

In case TESTAR discovers a state s that contains a *form* widget (line 4), TESTAR obtains the *form* widget (line 5) to only derive the *formActions* that can be performed in the child widgets of the form (line 6). These child widgets include all form elements like dropdown, checkbox, etc., and not only inputs. When all *formActions* are derived, TESTAR relies on the form-filling ASR to select the next action to execute (line 7).

Algorithm Form_filling_ASM

Require: State s ▷ The current state of the SUT
Require: ASR ▷ The grammar-based form-filling ASR

- 1: **if** s does not contain any *form* widget **then**
- 2: $allActions \leftarrow$ derive all actions in state s
- 3: $a \leftarrow$ select randomly from *allActions*
- 4: **else** ▷ The state contains a form
- 5: $form \leftarrow$ obtain form widget of s
- 6: $formActions \leftarrow$ derive *form* filling actions in s
- 7: $a \leftarrow$ select from *formActions* with ASR
- 8: **end if**
- 9: **return** a ▷ Return the selected action

InputDataManager

When testing, input data is required to fill text fields. To ensure the use of correct data values (e.g., numeric, email, dates, etc.), a predefined pool of appropriate data values can be established for existing text-field widgets. This extends beyond ensuring correct types (e.g., numeric, text) or proper formatting (e.g., email, dates). For instance, bank applications require the use of existing account numbers to successfully submit transactions, while market applications necessitate searching for existing stock products.

The InputDataManager class in TESTAR manages input data generation for actions that involve typing text in fields. Depending on the web field type, it generates values that comply with the field's specific constraints.

3.2.3 Empirical Evaluation

The study follows the Wohlin et al. experimentation guidelines [297] to evaluate the effectiveness and efficiency of using grammar in the generation of test sequences. To achieve this, the

proposed grammar for action selection rules is compared with random action selection. The grammar-based action selection is referred to as the human-like strategy because it allows TESTAR to mimic the testing strategies typically employed by humans.

The following research questions are formulated for this study:

- Grammar-RQ1: Is the human-like strategy more effective in filling out forms compared to random?
- Grammar-RQ2: Is the human-like strategy more efficient in filling out forms compared to random?

To analyze the results and answer the research questions, the following hypotheses are defined:

- Grammar-HO_A: The human-like ASR does not have a higher success rate than random.
- Grammar-HO_B: The human-like ASR does not have more uniform widget interactions than random.

This empirical study was conducted using data from two experiments. In each experiment, a different object was used: web forms generated with WebformSUT and the Parabank website.

SUT objects

To be able to concentrate on form-filling only, and not be distracted by other components that can appear in the same state as a form, a web form generator was created: WebformSUT¹. It enables the creation of arbitrary web forms. To call up a web form, all that is required is to type in a URL that instructs the WebformSUT which fields the form will have and in which order. The WebformSUT constitutes the first SUT object used in the experiment.

Parabank [211] is a demo website developed by the Parasoft company to demonstrate web services and other software testing functionality. This demo website mimics a banking application and provides bank services to realize and search financial transactions like bill payments. By evaluating the grammar-based ASR on the Parabank website, the study aims to assess its ability to handle realistic web scenarios and its potential to replicate the testing strategies typically employed by humans in such applications. Parabank is the second SUT object used in the experiment.

¹WebformSUT is open-source available on the TESTAR GitHub repository - <https://github.com/TESTARtool/webformsut>

No test oracles were set for either SUT object. The *blocking principle* [297] was applied to disable TESTAR oracles to report exceptions, ensuring that the experiment focuses on the effectiveness and efficiency of form-filling.

Independent variables

The following independent variables were used in the experiments:

- The TESTAR ASR used is either random or the human-like ASR from Listing 3.2, with their corresponding protocols.
- In both SUT objects, an additional `KeyScrollDown` action is derived when widgets of the form are not visible below.

There are independent variables that correspond to additional configuration for Webform-SUT:

- The *abstraction strategy* aimed to track which form-filling actions were executed and which remained unvisited in the WebformSUT:
 - As the *main abstraction mechanism*, TESTAR used the `WebId` property to identify the widgets and the state. Then, the `OriginWidget`, `OriginState`, and `ActionRole` were used to identify the actions performed.
 - As custom *abstraction sub-strategy*, TESTAR was programmatically configured to identify all radio button actions in a common group as the same action by using the `WebName` property and excluding the web id property. This enables TESTAR to only interact with one radio button widget per group.
- The `InputDatamanager` was used for some of the fields (email, url) that require that the input matches their format, limits range (number, range), or demands correct values (date, time, datetime), but there are no restrictions otherwise. None of the fields are mandatory.
- Action duration was set to the default 0.1 seconds.
- The time to wait between actions was set to the default 0.1 seconds.
- Every run consists of 1 sequence of 1.5 times the max group form size. For short forms, this is 45, and for long, 105 actions. This represents the upper limit, as the run will stop upon submitting.

- Each group has 30 forms.

Moreover, independent variables correspond to additional configuration for Parabank:

- The algorithm `Form_filling_ASM` is enabled.
- The *abstraction strategy* aimed to track which form-filling actions were executed and which remained unvisited in the Parabank SUT:
 - As the *main abstraction mechanism*, TESTAR used the `WebId` and `WebName` properties to identify the widgets and the state. Then, the `OriginWidget`, `OriginState`, and `ActionRole` were used to identify the actions performed.
 - As custom *abstraction sub-strategy*, TESTAR was programmatically configured to include the `WebURL` of the state for creating the action identifiers of the `KeyScrollDown` action. In comparison with `WebformSUT`, during a scriptless exploration in Parabank, TESTAR can find multiple forms with scroll-down options, and it is necessary to distinguish these actions depending on the form URL.
- The `InputDatamanager` was used for amount fields that require numeric values, date fields valid dates with a specific format, and account fields that require existing Parabank account numbers. Thus, a *required unsuccess* submission only happened if the action selection strategy did not select the form widget to be filled, but not because the data value was inappropriate.
- Action duration was set to 0.5 seconds.
- The default time to wait between actions was set to 1 second, except in the case of a submit button, where it was set to 5 seconds. This was because Parabank requires more time to transit to the next state after a submit.
- Every run consists of 1 long sequence of 500 actions. In Parabank pre-executions, it was registered that 500 actions were sufficient to allow random exploration to find all the existing forms.
- A correct login was forced each time TESTAR connects with the SUT.
- The admin and logout buttons were filtered to allow ASR to maintain the exploration on the main bank states of Parabank. Additionally, the dynamic links created in run-time providing access to an activity bank state were filtered to reduce the exploration space.

Dependent variables

To evaluate the effectiveness and efficiency of filling forms in both WebformSUT and in Parabank, the following dependent variables were measured for each SUT and test run:

- The total number of (un)successful tries to submit a form.
- The rates of interaction with the form widgets.

To measure effectiveness, the evaluation considered how often forms are successfully submitted compared to the total number of submission attempts. For WebformSUT, each run was allotted a maximum number of actions, and failing to submit before all actions had been spent counted as an unsuccess. Parabank's forms have mandatory and required fields, which adds the additional requirement that said fields must be adequately dealt with before submitting.

Efficiency was measured through how much the strategy interacts with the widgets. Ideally, all widgets are visited exactly once. Neither over-interaction (revisiting the same widgets) nor under-interaction (skipping some of the widgets) is desirable. The former lowers efficiency, for it takes up more time than needed to handle the form; while the latter leads to incomplete testing data, and may even cause the submit attempt to fail.

Note that both experiments handle visiting the forms differently. Each generated form was run 30 times, and success or not is dependent on whether the form is submitted before TESTAR has used up all allotted actions. In contrast, during the exploration of Parabank, TESTAR can visit the states that contain the different forms multiple times in the same run.

Experiment operation with WebformSUT

WebformSUT can generate forms with the following W3² input types: *button, checkbox, color, date, datetime-local, email, month, number, password, radio, range, reset, search, submit, tel, text, time, url, week*. Each of the field types has an associated three-character code, such as *ema* for email, *tel* for telephone, etc.³. The URL's format is `http://webformsut.testar.org/forms/xxx` where *xxx* is a string concatenation of the codes that represent the fields.

For example, the URL `http://webformsut.testar.org/forms/emada1timtel` produces a form with the following fields: email, date, time, telephone. The WebformSUT will thus provide a single form with these four fields in the specified order shown in Figure 3.19.

²https://www.w3schools.com/html/html_form_input_types.asp

³Complete list can be found here: <https://github.com/TESTARtool/webformsut/blob/main/forms/views.py>

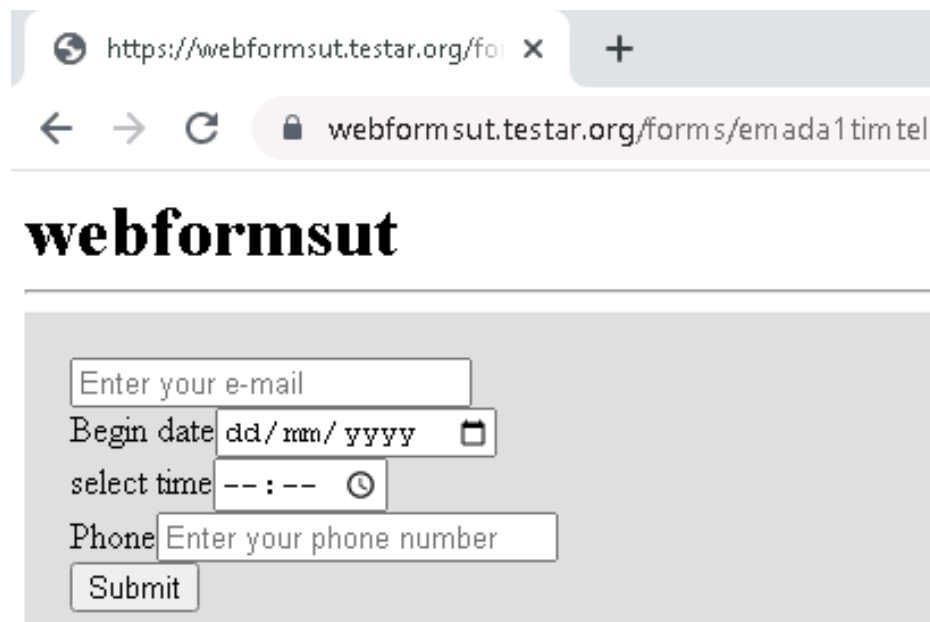
A screenshot of a web browser displaying a form on the website 'webformsut.testar.org'. The browser's address bar shows the URL 'https://webformsut.testar.org/forms/emada1timtel'. The page title is 'webformsut'. The form itself is a light gray box containing several input fields: a text field for 'Enter your e-mail', a date field for 'Begin date' with a calendar icon and the format 'dd/mm/yyyy', a time selection field for 'select time' with a clock icon and the format '--:--', and a text field for 'Phone' with the placeholder 'Enter your phone number'. A 'Submit' button is located at the bottom of the form.

Fig. 3.19 An example of a WebformSUT form

Two web form lengths (short and long) and two web form types (text-only and mixed) were generated, resulting in four groups of forms in total.

Each group needs enough variety to serve as a comparison between the random and human-like strategies. Therefore, 30 forms were generated per group. Furthermore, to deal with the randomness and to be able to obtain valid conclusions about the possible rejection of the null hypotheses, each form was run 30 times per ASR (random and human-like) [18]. Consequently, the experiment consisted of executing $4 * 30 * 30 * 2 = 7.200$ trials to compare the random and human-like form-filling ASR from Listing 3.2. Each trial measured variables related to the effectiveness of trying to succeed when submitting a form and the efficiency of filling all form widgets once before submitting the form.

Fields may be present more than once, and form lengths fall within a range depending on their group. Short forms (10-30 fields) fit onto the screen in their entirety, while long forms (50-70 fields) need at least one page-down scroll.

The division between text-only and mixed fields aims to mimic the setup of real-life web forms. Signing up for an account, newsletter, or other service corresponds to web forms that offer mainly text fields to fill in the necessary information. Other types of web forms, such as filtering options for web shops or settings in digital environments, are usually a mix of field types.

Through batch files, 4 Windows Virtual Machines (VM) with 8 CPU cores and 16 GB RAM were configured to each run a part of the experiments (see Figure 3.20). Installing a

clean copy of TESTAR beforehand ensures that no artifacts of previous runs were left behind. Each VM collects the data of its own runs, and these are brought together on the centralized storage server, from where they are retrieved for analysis.

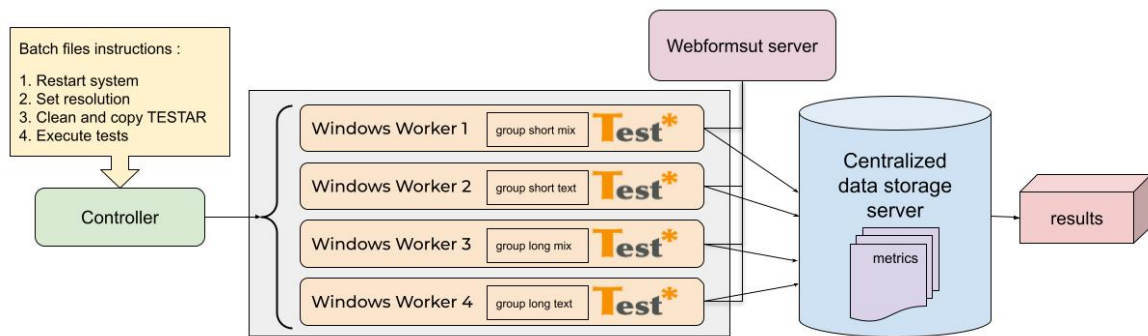


Fig. 3.20 Architecture of WebformSUT experiment

To deal with the randomness and to be able to obtain valid conclusions about the possible rejection of the null hypotheses, we repeat the execution of the experiment 30 times

Experiment operation with Parabank

Initial experiments provide valuable insights into the effectiveness and efficiency of the grammar-based approach. However, forms can comprise a wider variety than input elements, such as text areas or select dropdowns. Also, they can be integrated into complex states containing forms composed of multiple submit buttons besides other non-form GUI elements. Thus, the evaluation continued with an experiment to test the human-like ASR in a more realistic setting.

Parabank provides a realistic example of the varied characteristics that web forms can have. Depending on the specific objectives of the form, the existing form elements can have empty or pre-populated values that require optional or mandatory filling actions. Additionally, composed forms may be subdivided into multiple sections, allowing users to select and fill partial and desired fields.

Figure 3.21 shows an example of 2 Parabank forms with different characteristics. The left customer care form contains a set of mandatory inputs that must be filled to permit submission. The right find transactions form is subdivided into 4 possible searching groups, with mandatory correct values input to succeed in the search submission. It also has a pre-populated select dropdown that can be optional to interact with.

The Parabank form widgets were divided into three different categories:

Fig. 3.21 Parabank customer care and find transactions forms

- **pre-populated:** form widgets that already contain a predefined value and, therefore, are optional to interact and be filled.
- **mandatory:** empty form widgets that must be filled so that the forms allow submission. If these widgets are empty at a submit, the form does not allow to continue until the value is filled.
- **required:** empty form elements that need to be filled so that the forms submit successfully. If these widgets are empty at submit, the form transits to an erroneous state.

Whereas the *pre-populated* form widgets do not require interaction to succeed in the form submission, the *mandatory* and *required* form widgets can provoke two types of unsucceeded submissions:

- **mandatory unsuccess:** trying to submit the form does not change the form state. The SUT forces the user to fill in widgets properly. TESTAR will remain filling the form until all form widgets are filled properly.
- **required unsuccess:** submitting the form transits to a new erroneous state. Some widget that requires filling in an adequate value remained empty or with an inappropriate value when submitted. TESTAR can submit the form with inappropriate values and not trigger the form correctly.

In Parabank, ignoring the filtered logout, admin, and activity states, there were seven distinct web forms with various form widget categories:

- **openaccount:** 2 prepopulated select elements and 1 submit button. This form will always *succeed*.
- **updateprofile:** 7 prepopulated text inputs and 1 submit button. This form will always *succeed*.
- **transfer:** 1 mandatory numerical text input, 2 prepopulated select elements, and 1 submit button. This form can provoke a *mandatory unsuccess*.
- **customercare:** 4 mandatory text inputs and 1 submit button. This form can provoke a *mandatory unsuccess*.
- **billpay:** 9 mandatory text inputs, 1 pre-populated select element, and 1 submit button. This form can provoke a *mandatory unsuccess*.
- **findtrans:** 1 pre-populated generic select element and 4 searching groups containing 1 or 2 mandatory inputs. Also, each group has an associated submit button. This form can provoke a *mandatory unsuccess*.
- **requestloan:** 2 required numerical text input, 1 pre-populated select element, and 1 submit button. This form can provoke a *required unsuccess*.

The experiment consists of executing two sets of trials with Parabank to compare the random and human-like ASR from Listing 3.2. As with WebformSUT, each ASR (random and human-like) was executed 30 times to deal with the randomness and to be able to obtain valid conclusions about the possible rejection of the null hypotheses [18].

Random and human-like strategies, combined with the form-filling action selection mechanism, were compared. Each strategy was executed 30 times in a single Windows VM with 8 CPU cores and 16 GB RAM. Figure 3.22 shows how the multiple runs were executed.

After preparing the resolution of the VM and the TESTAR protocol with the corresponding strategy, TESTAR invokes a batch file that stops any Apache web server still running, cleans the last deployed Parabank instance, and starts a new Apache server which deploys a clean instance of Parabank. This ensures that each run begins in the same initial web state of a SUT with identical web elements. Finally, the results of the sequences and the metrics that measure the dependent variables data are created in the output directory of TESTAR.

Developing and validating the TESTAR protocol and obtaining metrics within the designed architecture required approximately 20 hours of manual effort.

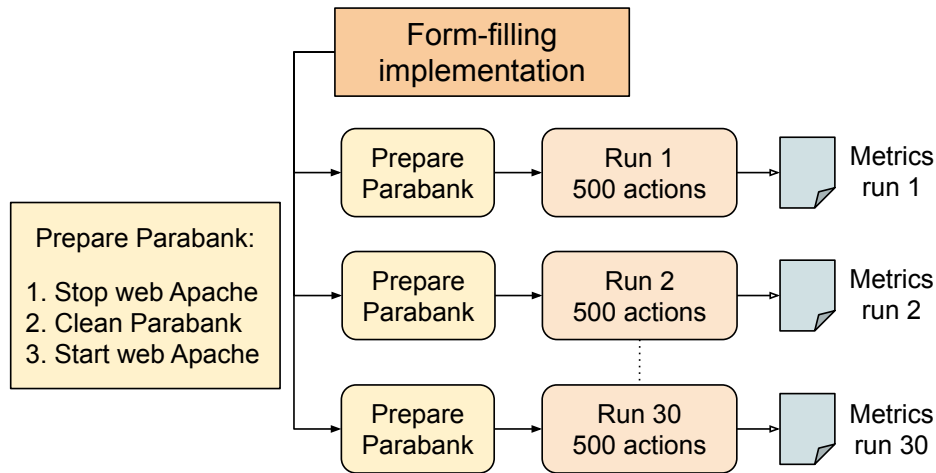


Fig. 3.22 Architecture of Parabank experiment

3.2.4 Results

The following sub-sections present the results obtained for the WebformsSUT and the Parabank SUT, separately. After that, the answers to the research questions and the possible threats to validity are discussed.

WebformsSUT Results

Table 3.3 Number of p-values < 0.05 for WebformsSUT.

	actions executed	success
short mix	5 / 30	12 / 30
short text	6 / 30	21 / 30
long mix	11 / 30	22 / 30
long text	21 / 30	30 / 30
	Mann Whitney U test	Two Proportion Z-Test

Due to the large number of p-values (3 times 120 individual values), the outcome of the statistical Mann-Whitney U test and Two Proportion Z-Test were summarized in Table 3.3. The full results are available online⁴. As the p-values indicate, the human-like ASR does not significantly differ from random in how many actions it takes for form-filling. However, it does significantly impact the rate of successful submits, particularly for longer forms.

Figure 3.23 shows the success rates of both approaches across the four possible groups of WebformsSUT forms. 900 successful submits is the maximum possible and represents

⁴<https://doi.org/10.5281/zenodo.10143346>

a 100% success rate, which is represented in the y-axis of Figure 3.23. Across all groups, the human-like ASR achieves consistently higher success rates of 95.9% on average for all runs, compared to random's very varied performance with an average of 57%. (Notably, the human-like ASR has a perfect 100% in the long text group.) This is most prominent in the longer forms, likely because more fields mean more options, and thus avoiding already-visited actions becomes both more important as well as harder to accomplish. Random has a tendency to repeatedly fill in the same fields while leaving others unvisited.

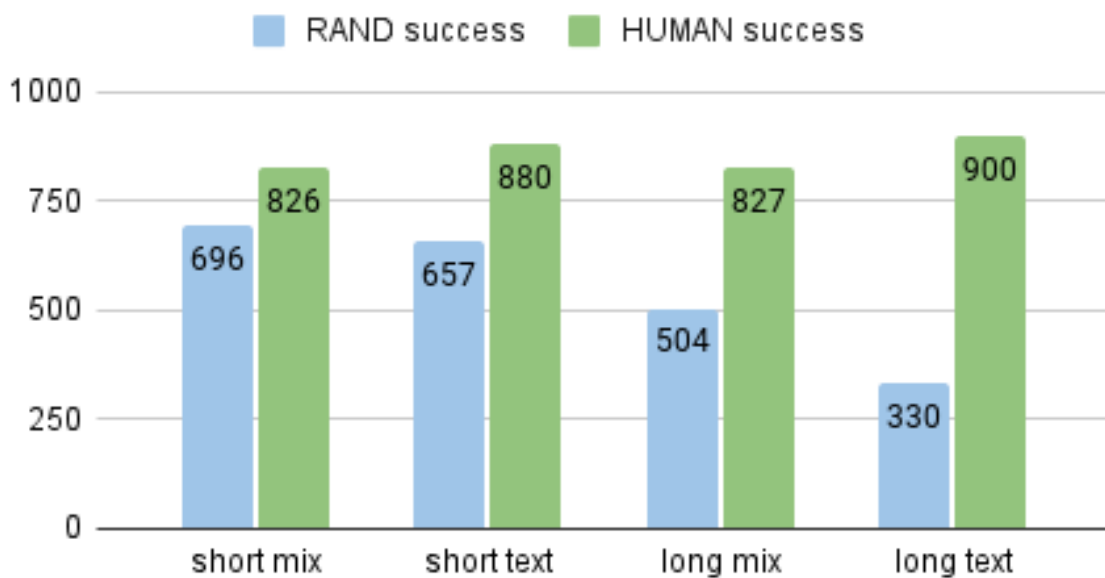


Fig. 3.23 WebformsSUT - success rates

Figure 3.24 shows the deviation of the action-field ratio per group of experiments for both approaches. The zero line represents exactly one interaction per field. As the graph shows, the human-like ASR performs close to the ideal for all three groups (0.0833 to 0.04), while the random ASR either under- or over-interacts with the fields (-0.6364 to 1.5).

In conclusion, we can reject both null hypotheses and conclude that the human-like ASR is more effective and efficient in form-filling than random in WebformsSUT.

Parabank Results

The statistical Mann-Whitney U test was performed to obtain evidence that human-like ASR is more efficient and effective than random for form-filling. The dependent variables of each of the seven Parabank forms were analyzed statistically. The results of p-values presented in Table 3.4 demonstrate that the human-like ASR TESTAR is significantly better than random for all seven forms of Parabank. Thus, we can reject the null hypotheses and conclude that

Webformsut Actions Per Field Deviation



Fig. 3.24 WebformsSUT - actions per field deviation

the human-like form-filling ASR is more effective and efficient in filling the form widgets compared to the random mechanism in Parabank.

Table 3.4 Mann-Whitney U p-value significant difference

Submitted coverage per form	Success	Unsuccess	Deviation
openaccount	p=3.0244e-05	—————	p=8.0437e-12
updateprofile	p=0.00011472	—————	p=1.8660e-11
transfer	p=0.00019728	p=1.8438e-10	p=1.1970e-12
customer care	p=0.00018417	p=1.6944e-12	p=4.0987e-12
billpay	p=2.4355e-08	p=2.2300e-12	p=2.1431e-10
findtrans	p=1.9992e-06	p=4.3795e-12	p=1.1691e-11
requestloan	p=3.4495e-11	p=5.2972e-11	p=1.2063e-11

Figure 3.25 shows the successful and unsuccessful submits that the random and human-like form-filling ASR accomplish for the customer care form (left form shown in Figure 3.21). The figures for the other six Parabank forms are available online ⁵. As Figure 3.25 shows, in the 30 total runs, the human-like ASR performed more than 60% successful customer care

⁵<https://doi.org/10.5281/zenodo.10143346>

submits than the random mechanism. The difference with the unsuccessful submits is even more significant. The random mechanism performed more than 300 unsuccessful tries, while the human-like ASR only made one unsuccessful attempt on this Parabank form. Also, after analyzing this unsuccessful attempt, it was discovered that the human-like ASR correctly selected the GUI action. It was the TESTAR tool that failed when performing the GUI mouse and keyboard events, trying to add the value in the selected text field.

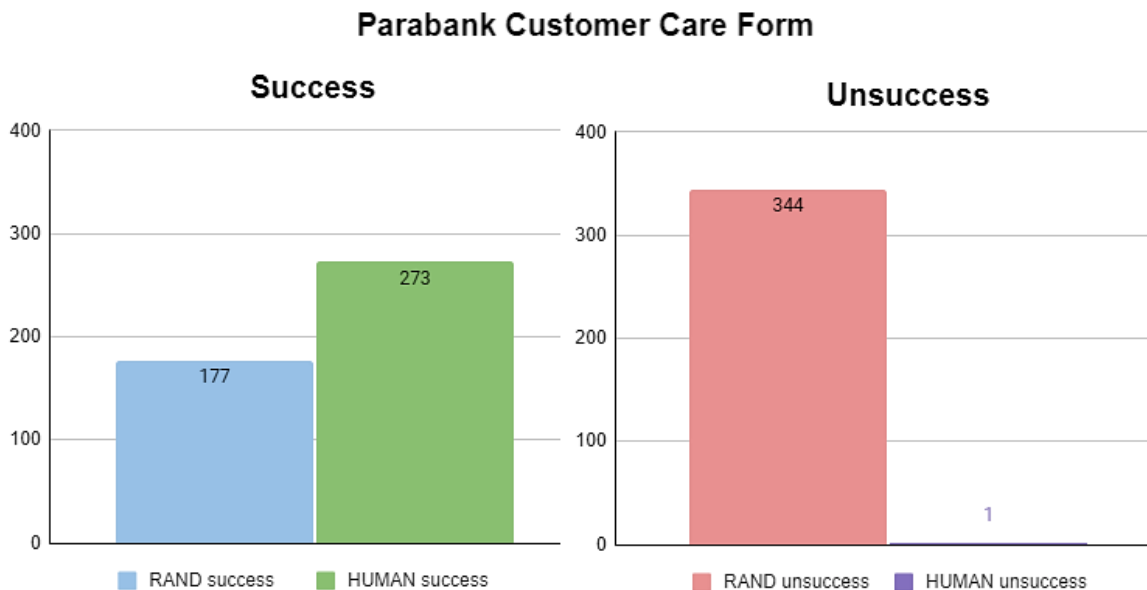


Fig. 3.25 Customer care form - success and unsuccess submits

Figure 3.26 shows the over-interacted deviation of the random mechanism when interacting with the Parabank customer care form. The columns represent the total number of successful submits in each TESTAR run. The lines of phone, name, message, and email form widgets represent the total interactions performed. Note that the setup of the experiment with Parabank, with known forms and limited fields, allows for a more detailed graph with a breakdown of interactions per field.

For Parabank as a whole, the average success rates were 99.8% (human-like ASR) and 55.7% (random ASR). The average widget interaction deviations were 0.0181 (human-like ASR) and 0.7774 (random ASR).

Figure 3.27 shows the uniform distribution of interacted form widgets when using the human-like ASR mechanism to fill the Parabank customer care form. In this case, the multiple runs only show a deviation of one action in run 9 when comparing the number of successful submits and the interaction with the form widgets. Along all the Parabank forms filled with the human-like ASR, only one run contains an outlier in the deviation when trying

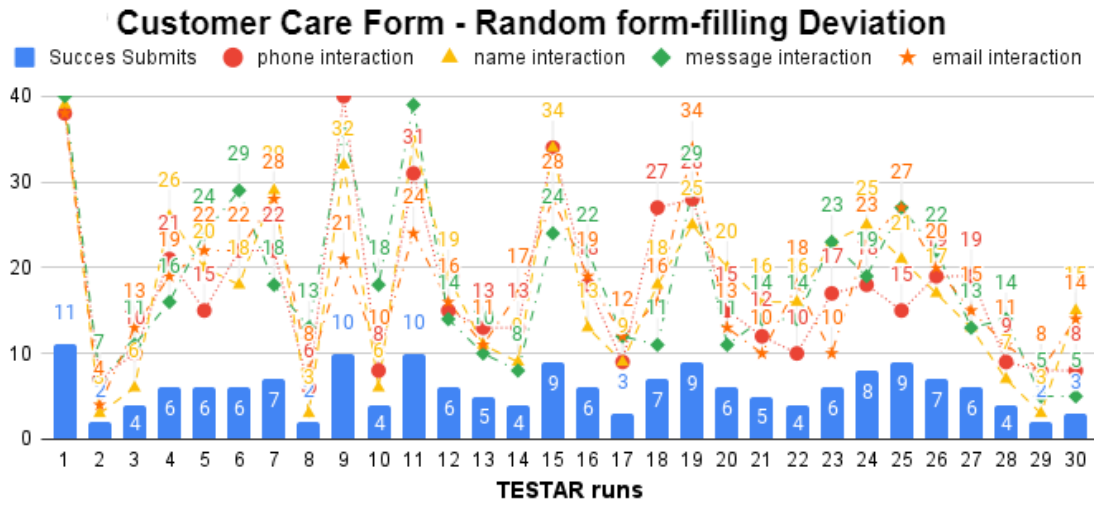


Fig. 3.26 Customer care form - random form-filling deviation

to scroll down in the bill payment form. Despite this, human-like ASR is significantly better than random.

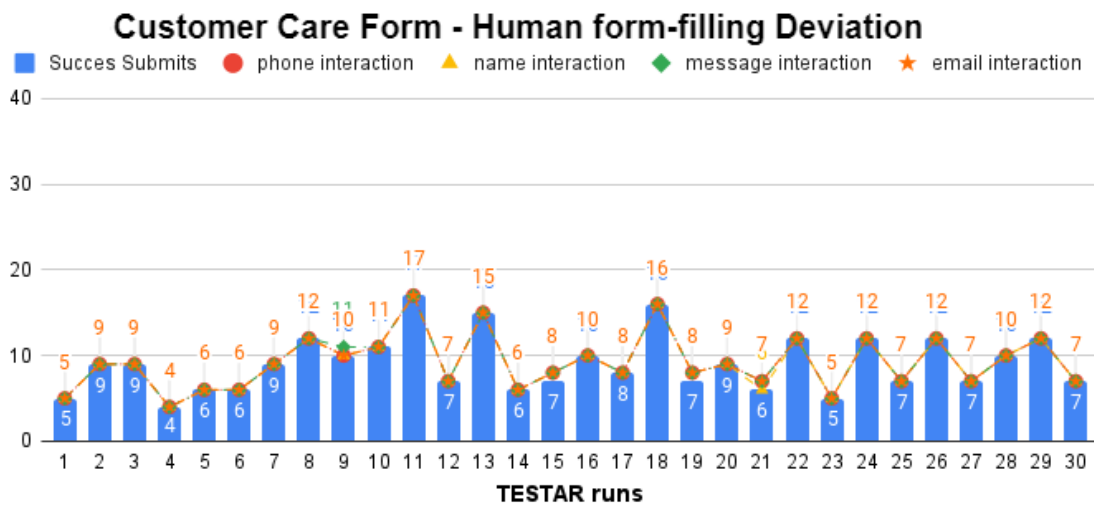


Fig. 3.27 Customer care form - human-like form-filling deviation

Answer to RQs

We can reject the null hypothesis Grammar-HO_A: ("The human-like ASR does not have a higher success rate than random") because both WebformsSUT and Parabank experiments have shown that the human-like ASR has a higher number of successful submits and fewer failures than random. This translates to a significantly higher success rate for the human-like ASR.

Therefore, for Grammar-RQ1: *Is the human-like strategy more effective in filling out forms compared to random?*, we observe that it takes the human-like ASR fewer tries to submit the same form as random. Moreover, Parabank's mandatory and required forms are hard to navigate by the random ASR.

The next null hypothesis Grammar-HO_B: ("The human-like ASR does not have more uniform widget interactions than random") can also be rejected. In contrast to random's varying levels of widget interaction, the human-like ASR interaction rate is very stable throughout all runs in both experiments.

Similarly, for Grammar-RQ2: *Is the human-like strategy more efficient in filling out forms compared to random?*, we observe that the human-like ASR is able to accomplish more or equal successful submits with the same number of actions. This is clearly seen in the results of the Parabank experiment, where, on average, the human-like ASR made more submits per run.

3.2.5 Threats to validity

This section presents some potential threats that may impact the validity of the results [297, 232, 258].

Construct validity

Construct validity is a crucial aspect of empirical studies, as it ensures the results' reliability. We pay special attention to ensuring construct validity by incorporating several measures. We consider **face validity** by clearly defining the experimental design and identifying the metrics required to address our research questions. Additionally, we focus on **content validity** by utilizing observable metrics at the GUI level, such as the number of web forms completed and submitted, in line with other studies in state of the art, such as [65].

Internal validity

In terms of internal validity, we acknowledge that both strategies rely on a random exploration of the SUT to detect web forms, which could potentially result in certain forms being missed by the testing tool and, thus, not filled or submitted. To mitigate this threat, we have conducted pre-experiments to evaluate the exploring capacity of the grammar-based rules. Moreover, specifically for Parabank, we have conducted pre-experiments consisting of 500 actions per run, and we have executed these experiments 30 times to ensure that the process can visit the seven forms in the SUT.

Additionally, we are aware that the size of both SUTs could be a confounding factor in our experiments. Thus, we will run more experiments with bigger SUTs in future experiments.

Conclusion validity

We used parametric and non-parametric tests depending on the normality of data to assure conclusion validity. Furthermore, we set the alpha level at 0.05, a commonly used threshold for determining statistical significance.

External validity

With regard to external validity, we have focused on web applications and a specific challenging task, namely the automated filling of web forms. Although the grammar-based action selection rules we propose in this paper are generally applicable to other testing tasks since they simulate human behavior, we specifically chose the web form filling since it is a common feature in almost all web applications and it also serves as the entry point to the hidden web. Nevertheless, we are aware that we require more experiments with different types of SUTs, such as mobile, XR [215, 226] or other types of systems, and take different testing challenges (such as scrolling or resizing GUIs) to improve the generalization of our results.

3.3 Scriptless exploration of game environments

The scriptless testing approach used by TESTAR automatically generates test sequences at run-time to explore a System Under Test (SUT) by selecting and executing the available actions in the discovered states. While this approach appears well-suited for 3D games, existing scriptless testing tools are primarily designed for desktop, web, and mobile GUI applications. Adapting these techniques for game testing requires addressing distinctive 3D game features, such as precise position and orientation data for character movements and properties of interactive elements.

As presented in previous Chapter 2.2, TESTAR leverages the IV4XR framework [223] to bridge the gap between current scriptless testing tools and technologies capable of discerning a game's states. The integration of the iv4XR LabRecruits-plugin enables the testing tool TESTAR to explore 3D games autonomously.

Although this experimental game lacks a specific testing cycle, it was created to evaluate sophisticated decision-making algorithms to explore 3D games autonomously. Therefore,

in this section, we present an ASM that enables the TESTAR *agent* to improve the spatial exploration effectiveness with the LabRecruits game.

3.3.1 TESTAR for the LabRecruits experimental 3D game

Using the LabRecruits-plugin, the TESTAR *agent* can perform the scriptless operational flow shown in Figure 3.28 to test the experimental 3D game.

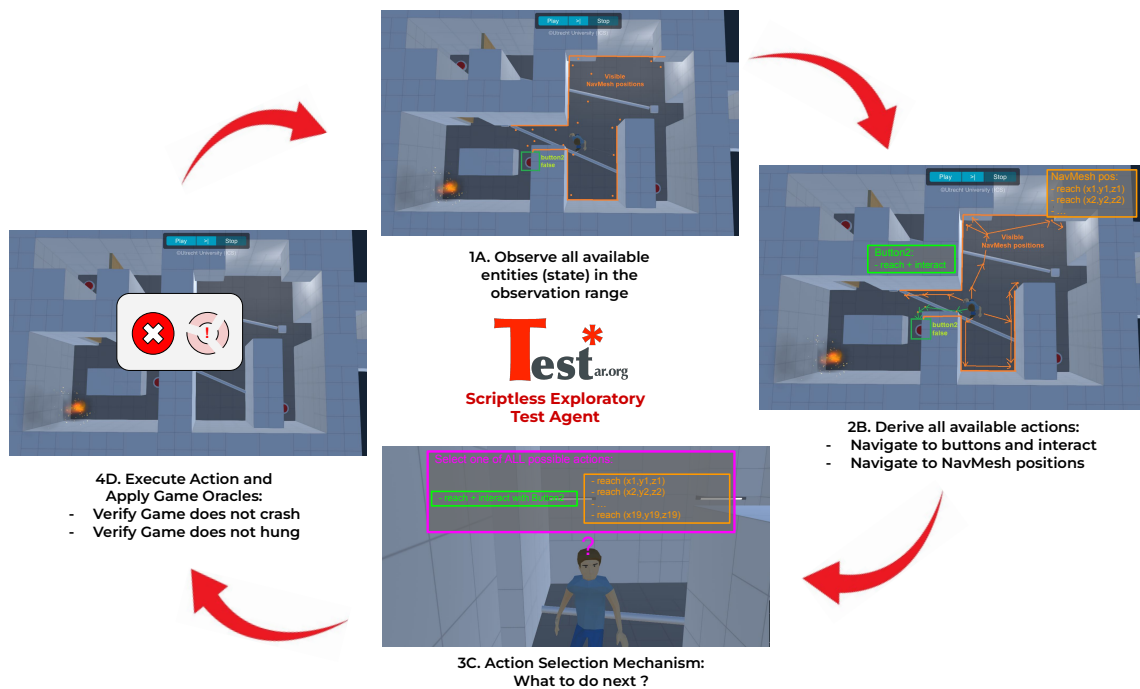


Fig. 3.28 TESTAR operational flow with LabRecruits

TESTAR agent: Game state in LabRecruits

The TESTAR *agent* employs the LabRecruits-plugin to actively *Observe* all the game entities that reside in the observation range area. Each game entity contains a set of aforementioned properties, such as position and type for all entities and health and score for the *Character* entity. Together, the observed entities with their properties constitute the game *state*.

TESTAR agent: Derived actions and navigation in LabRecruits

Depending on the type of entity, the TESTAR *agent* derives tactical actions to reach and interact with the entities. For example, the *agent* may be interested in interacting with buttons but not with fire.

A distinctive characteristic between testing traditional GUI software and games is the need to reach the desired entity to interact, as well as explore through states where no functional entities exist in the observed area to potentially discover new game entities. While the LabRecruits-plugin's *Navigation Mesh* (NavMesh) and the A* pathfinding algorithm provided by the IV4XR framework facilitate finding the optimal movement path between initial and destination nodes, there is a necessary decision-making step at the top level to determine which action to derive and select during the exploration process.

To reach the desired entity to interact, the TESTAR *agent* exploits the *navigation* capabilities of the LabRecruits-plugin to calculate the *navigable* path of positions to reach the target entity. Moreover, to potentially discover new game entities, the TESTAR *agent* not only considers deriving actions that interact with observed entities but also derives actions that explore visible *navigation* positions. To do this, TESTAR's protocol has been extended so that after deriving all available interaction actions with visible entities, it also derives all available exploration actions to visible positions.

TESTAR agent: Action Selection Mechanism in LabRecruits

After deriving all available actions, the TESTAR *agent* uses, by default, a random ASM to decide which action to execute next. Although random ASMs have proven practical for traditional software [276], for exploring 3D games, it is necessary to research more sophisticated ASMs.

Let us consider the example in Figure 3.28. First, the TESTAR *agent* observes one button and derives one action to navigate and interact with it. Second, there are 19 positions in the observation area (i.e., 19 orange navigable dots). Hence, the TESTAR *agent* derives other 19 available exploration actions.

A random ASM will have around 5% probability of selecting the unique interaction actions from the 20 total actions. This increases the chance of selecting an exploration action to around 95%. Moreover, within the set of available exploration actions, selecting remote positions that remain unexplored can potentially allow the TESTAR *agent* the discovery of new entities. The Algorithm **LabRecruits_Interactive_Explorer_ASM** was developed to enhance the exploration of unexplored positions.

The LabRecruits_Interactive_Explorer_ASM tracks a list of *interacted* entities and a list containing the *explored* NavMesh positions. First, the ASM checks whether the set of available *actions* contains an action that interacts with a non-interacted entity (line 1). In that case, because there can be several non-interacted entities, it prioritizes choosing the nearest entity (*nearEntity*) to the *agent* (line 2). Thus, the ASM selects the action that navigates

Algorithm LabRecruits_Interactive_Explorer_ASM

Require: *interacted* ▷ List the interacted entities
Require: *explored* ▷ List of explored NavMesh position
Require: *actions* ▷ All available state-actions

- 1: **if** *actions* contains entities that were not *interacted* **then**
- 2: *nearEntity* ← nearestEntity(*actions*)
- 3: *a* ← select to navigate and interact with the *nearEntity*
- 4: **save** *nearEntity* as *interacted*
- 5: **else if** *actions* contains positions that were not *explored* **then**
- 6: *remotePos* ← remotePosition(*actions*)
- 7: *a* ← select to navigate to explore the *remotePos*
- 8: **save** *remotePos* as *explored*
- 9: **else**
- 10: *a* ← random selection from all actions
- 11: **end if**
- 12: **return** *a* ▷ Return the selected action

and interacts with the *nearEntity* (line 3), saves this *nearEntity* as interacted to not to be prioritized in the next iterations (line 4), and finally, returns the selected action (line 12).

Second, the ASM checks whether the set of available *actions* contains an action that explores a position out of the *explored* NavMesh positions (line 5). If so, because there can be several unexplored positions, it prioritizes under a small threshold the choice of a remote position (*remotePos*) to the *agent* position (line 6). Consequently, the ASM selects the action that navigates and explores the *remotePos* (line 7) and includes the position in the *explored* list to enhance selecting other unexplored positions in the next iterations (line 8). Finally, the ASM returns the selected action (line 12). In case the actions do not contain a non-interacted entity or non-explored position (line 9), the ASM selects (line 10) and returns an action randomly (line 12).

TESTAR agent: Oracles in LabRecruits

TESTAR integrates generic oracles intended to verify the robustness of the SUT: detect if the process has crashed or hung or if the state elements, or debugging logs, contain suspicious exception messages. Although these oracles already helped in the initial versions of the IV4XR framework to detect an invalid action request that hangs the framework ⁶, this paper primarily focuses on evaluating the effectiveness of ASM exploration.

⁶<https://github.com/iv4xr-project/iv4xrDemo/issues/9>

3.3.2 LabRecruits empirical evaluation

In order to assess the efficacy of scriptless testing for exploring the LabRecruits game, we evaluate the potential benefits of investing time and effort in developing ASMs for more sophisticated exploration techniques. To accomplish this, we quantitatively measure the spatial coverage of discovered and interacted entities and navigated positions within a randomly generated scenario. To guide our study, we have formulated a research question and null hypothesis:

RQ: How effective is spatial exploration in the LabRecruits game when using different TESTAR ASMs?

H₀: The LabRecruits_Interactive_Explorer_ASM is not more effective than a random ASM in the LabRecruits game.

We designed a controlled experiment based on Wohlin's guidelines [297] and a methodological framework specifically built to evaluate software testing techniques [278].

The experiment consists of running the random default ASM and the more intelligent decision-making from Algorithm LabRecruits_Interactive_Explorer_ASM that prioritizes the interaction with newly discovered button entities and the exploration of remotely unexplored positions. Each trial measures spatial coverage, corresponding to the extent of interaction with discovered LabRecruits button entities and the exploration of unique NavMesh positions.

LabRecruits generated scenario

A randomly generated LabRecruits scenario was used to ensure an unbiased evaluation of the ASMs. The scenario features a 50x50 map with 1680 navigable positions and enclosed rooms that TESTAR must explore by discovering and interacting with buttons. The map is divided into 25 uniformly distributed 10x10 rooms. Each room contains floors and walls, with 24 interactive buttons randomly placed across these rooms. Doors are placed to connect adjacent rooms, and the *agent* starts in the first room, needing to explore the map.

LabRecruits independent variables

To focus on the exploratory capabilities of the ASMs and prevent the *agent* from dying, we prevent generating hazardous entities in the scenario. Since this study focuses on spatial exploration, we applied the *blocking principle* [297] to limit the TESTAR *agent* interactions with buttons that open closed doors. Moreover, the observation range through the LabRecruits-plugin is limited to encourage exploring and discovering new entities and positions.

LabRecruits dependent variables

To answer our research question, we measured the number of discovered and interacted buttons, as well as the observed and walked positions. The LabRecruits game allows for the instrumentation of the coverage of NavMesh positions, floor positions, and existing entities. This data enables real-time observations during the exploration process to obtain spatial coverage. Using this data, we can generate a 2D heat map highlighting the covered space.

Figure 3.29 shows an example of one exploratory sequence of 500 actions in the experimental map. The *agent*, which prioritized exploring remote positions, could cover almost all existing rooms in the generated scenario.

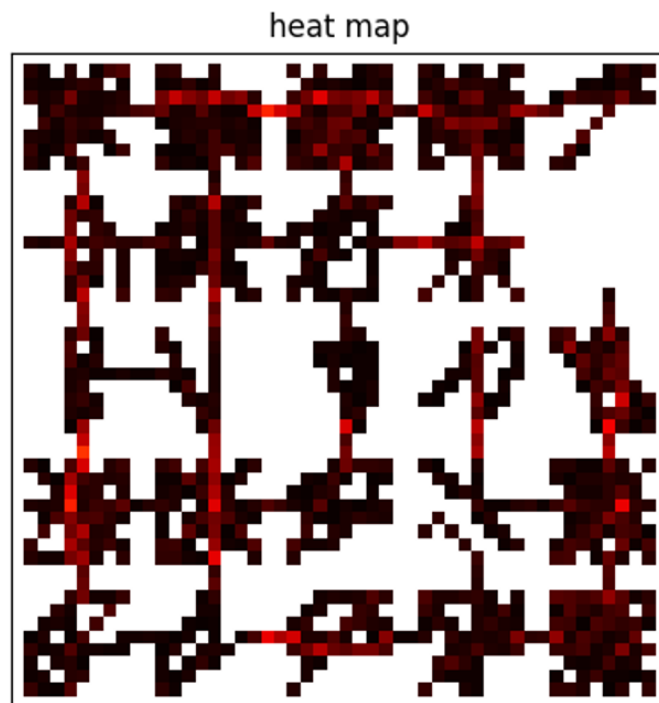


Fig. 3.29 LabRecruits spatial coverage heat map

LabRecruits design of the experiment

We evaluate the random and the LabRecruits_Interactive_Explorer_ASM by executing an exploration of 500 actions on the generated scenario. We repeated the exploration process 30 times for each ASM to obtain independent spatial coverage metrics for each execution. For each new execution, we reload the same initial scenario conditions in the same Windows machine with 2 CPU cores and 8 GB RAM.

3.3.3 LabRecruits Results

We first present the spatial coverage achieved in the 30 independent runs. Next, we use the Wilcoxon test to determine whether there is a significant difference between the ASMs. The experiments were performed in LabRecruits v2.3.3. The replication package can be found here ⁷.

Figure 3.30 shows the results for the observed and interacted blocks. Each line represents one of the 30 independent runs. The random ASM achieved a coverage ranging from 17% to 42% for observed buttons and 8% to 25% for total interacted buttons. In contrast, the LabRecruits_Interactive_Explorer_ASM achieved coverage ranging from 50% to 96% for observed buttons and the same 50% to 96% for total interacted buttons. This equivalence in percentages (i.e., 50% to 96%) results from the interactive explorer ASM prioritizing interaction with each newly observed button.

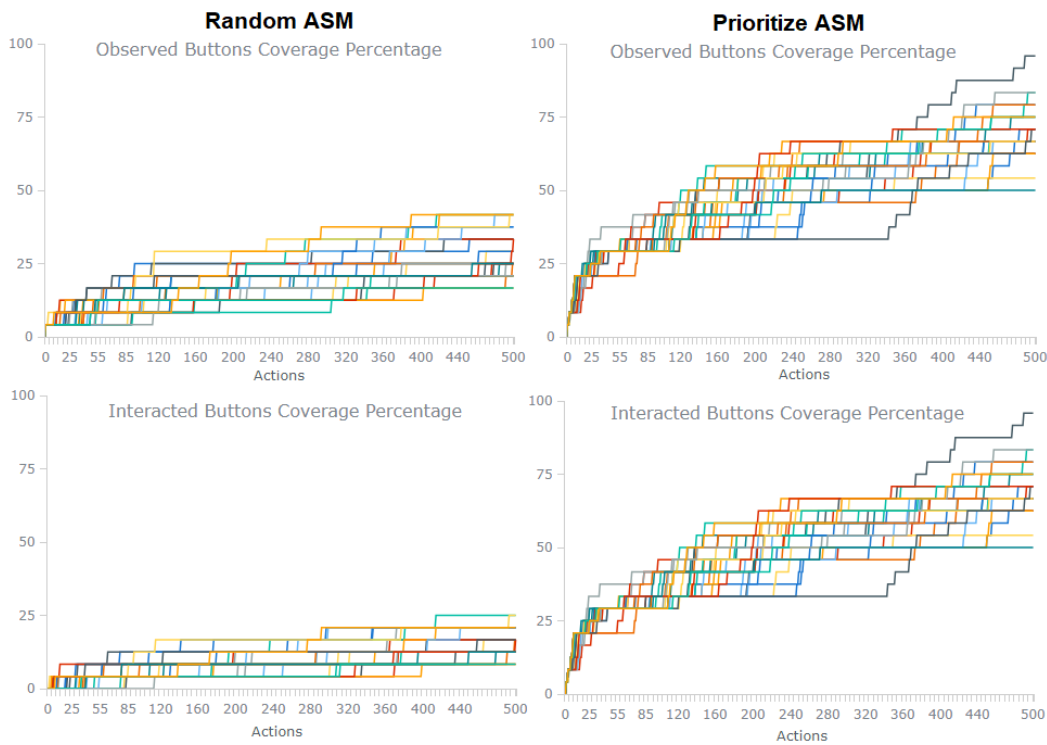


Fig. 3.30 Observed and interacted buttons coverage for LabRecruits

Figure 3.31 shows the results for the observed and walked floor positions. The random ASM achieved a coverage ranging from 6% to 17% for observed positions and 9% to 27% for walked positions. In comparison, the LabRecruits_Interactive_Explorer_ASM achieved a coverage ranging from 22% to 44% for observed positions and 36% to 60% for walked

⁷<https://drive.google.com/drive/folders/1yZiVERgyhx5iTsA8sL2QsBY36YQQoutT>

positions. In this LabRecruits scenario, the walked square positions are larger than the observed positions because the NavMesh positions are primarily created near geometric bodies such as walls and doors, rather than on empty surfaces like the floor.

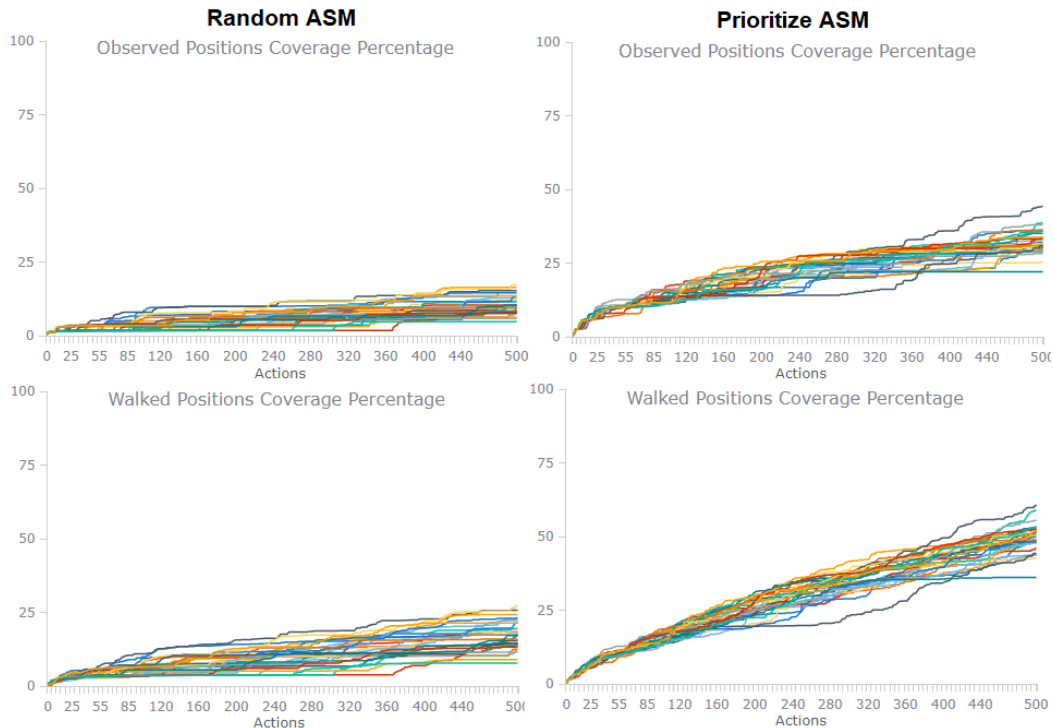


Fig. 3.31 Observed and walked floor positions coverage for LabRecruits

The LabRecruits_Interactive_Explorer_ASM outperforms the random ASM by prioritizing interacting with newly observed buttons and calculating efficient navigation paths to unexplored NavMesh positions. Table 3.5 shows Wilcoxon test results to verify a significant difference between the two ASMs for LabRecruits. We extracted values from the 30 different runs when executing 100, 300, and 500 actions. This means we calculate the significant difference in 3 different moments of the exploratory process. For the observed and interacted buttons and the observed and walked positions, the Wilcoxon test results show a p-value of less than 0.05, indicating that the LabRecruits_Interactive_Explorer_ASM is statistically superior to the random ASM. This allows us to reject H_0 for the LabRecruits experimental game and confirm that investing time and effort in developing intelligent ASMs benefits TESTAR exploration effectiveness.

Threats to validity

We discuss some threats to the validity of LabRecruits, according to [297, 232].

Table 3.5 Wilcoxon p-value significant difference for LabRecruits

Wilcoxon test p-values results for LabRecruits		
Executed actions	Observed buttons	Interacted buttons
100 actions	p=1.624e-06	p=1.331e-06
300 actions	p=1.666e-06	p=1.694e-06
500 actions	p=1.685e-06	p=1.672e-06
Executed actions	Observed Positions	Walked Positions
100 actions	p=1.733e-06	p=1.734e-06
300 actions	p=1.734e-06	p=1.733e-06
500 actions	p=1.733e-06	p=1.733e-06

Construct validity For the exploratory evaluation, we use the information from the LabRecruits scenario to design the concept of spatial coverage. Then, we use this data to measure the effectiveness of the random ASM and the LabRecruits_Interactive_Explorer_ASM. Although this spatial coverage is a self-design benchmark, the metrics come from the LabRecruits game's data.

Content validity For the exploratory evaluation, spatial coverage measures the existing buttons, NavMesh, and floor positions over a 2D scenario space. Still, there are various types of hazardous entities, and creating stairs to top rooms will require 3D spatial coverage. Although more sophisticated spatial coverage metrics can be researched in the future, the obtained 2D metrics allow us to measure the effectiveness of the exploratory ASMs.

Internal validity For the exploratory evaluation, we launched the LabRecruits scenario without hazardous entities to avoid the *agent* dying.

External validity The empirical study has been realized with the experimental game LabRecruits. Even though we demonstrated that the TESTAR *agent* has exploratory capabilities to effectively navigate LabRecruits automatically, we consider this to be a preliminary scriptless test automation evaluation.

Conclusion validity Due to the degree of randomness in the action selection of the exploratory ASMs, we cannot assume normal distribution in the experiments [18]. To address this, we repeated the exploration 30 times and used Wilcoxon statistical non-parametric tests on the results.

3.4 Summary

This chapter introduces various scriptless exploration strategies within TESTAR. The abstraction strategy used during the Software Under Test (SUT) exploration is crucial for distinguishing between discovered and unexplored states and actions. Without an adequate abstraction strategy that considers and deals with dynamism and non-determinism challenges, integrating intelligent Action Selection Mechanisms (ASMs) can become ineffective or even counterproductive.

The `State_Model_ASM` requires a stable abstraction strategy and memory from the state model inference process to identify and maintain a path leading to unvisited actions. Configuring a suitable level of abstraction in this process enhances the code coverage effectiveness of exploration.

These inferred models can be additionally used as artifacts for offline delta testing oracles (see Chapter 6). However, state model inference may be time-consuming for large SUTs, requiring further efficient approaches to reduce inference time (see Chapter 4).

Using grammar-based Action Selection Rules (ASR) in scriptless testing represents a novel approach that can complement traditional exploratory ASMs. By defining ASRs for form-filling mechanisms through this grammar, testers can replicate the testing strategies employed by human testers, resulting in more effective and efficient testing that more accurately reflects how the software is used by its users.

Lastly, the chapter introduces an interactive ASM designed to enhance the decision-making of the TESTAR *agent* when exploring an experimental 3D game. This scriptless interactive ASM motivated Keen Software House and GoodAI to adopt the TESTAR *agent* for testing the industrial game Space Engineers (see Chapter 7.5).

Chapter 4

TESTAR: Distributed exploration

Scriptless testing can benefit from state model inference to integrate more intelligent Action Selection Mechanisms (ASMs) (e.g., the unvisited or RL ASMs), defining offline oracles [73] or, as presented in Chapter 6, for comparing delta models of different versions of the same SUT to detect GUI changes [239]. However, for large and complex SUTs, the state model inference process can be time-consuming. Additionally, the inferred model can quickly become outdated in agile development environments, where multiple teams frequently update applications. The iterative and incremental software development processes necessitate quick and efficient state model inference to keep up with the constant changes.

Distributed system architectures enable multiple components or agents to work together to achieve a common goal. Implementing a distributed architecture for scriptless state model inference can allow multiple testing instances to collaboratively infer a shared state model, thus making the inference process faster and more scalable. However, when adopting a distributed architecture to reduce the time required for state model inference, it is essential to consider additional model inference challenges. These include dealing with the dynamic aspects and non-determinism in the GUI of the applications [195], handling state space explosion in the models [10], and the technical limitations of GUI testing tools that are often constrained to specific platforms, such as Windows desktop applications [169], Android apps [291], or web applications [181].

PATS framework [291] uses GUI ripping [175] for extracting GUI information from Android applications to generate distributed execution sequences on-the-fly coordinated by a central controller. However, PATS does not use the ripped information to build a GUI state model. Cartographer [44] proposes a distributed architecture with a central manager and parallel workers to infer a GUI model in a distributed way. Although they mention the importance of a suitable abstraction to infer a model, the paper only describes the approach briefly and does not contain any empirical evaluation. Crawljax [243, 181] constructs

a Document Object Model (DOM) state graph with click event transitions, using depth constraints to address state explosion, pattern comparators to filter dynamic elements, and defining crawl conditions to control some degree of non-determinism by checking whether a state should be visited. Crawljax demonstrated that increasing the number of distributed instances makes it possible to decrease up to 65% of the inference process. Nonetheless, Crawljax only considers mouse clicks and hover actions, which is a limited testing of an application from the user's perspective. Moreover, non-determinism can also be caused by the lack of information in the GUI, which may require inference algorithms that do not control but detect and adapt the GUI exploration algorithm.

Since the existing open-source tools for distributed GUI model inference show disadvantages, there was a need to evaluate the effort to extend other GUI model inference tools with support for distributed GUI exploration. AutoBlackTest [169] creates an abstract behavioral model of GUI states and actions by applying abstraction functions to widget properties while ignoring the inconstant properties of widgets. Although this tool allows the execution of click and type actions, it briefly acknowledges the existence of non-deterministic sequences without details about the possible impact on its action selection algorithm. Nevertheless, AutoBlackTest is not actively maintained and is restricted to desktop applications. Murphy tool [7] creates a graph to model the behavior of the GUI application using screenshot comparison, available GUI actions, and the abstraction of data values to define states. This tool supports many kinds of GUI interactions, mentions the state explosion challenge, and has been evaluated in a company. However, it does not detail how to deal with dynamism or non-determinism, and like AutoBlackTest, the tool has not been actively maintained.

Due to the limitations of the existing GUI model inference tools to support an architecture for distributed scriptless exploration, in this chapter, we decided to extend TESTAR to research the distributed approach. Section 4.1 presents a distributed state model inference approach that uses a set of strategies to deal with abstraction challenges and distinguish GUI states and transitions in the model. Then, Section 4.2 summarizes the distributed study results.

4.1 Distributed State Model Inference with TESTAR

In Chapter 3.1, the inference state model was presented for a single TESTAR instance. The unvisited actions first ASMs used the state model information to prioritize the selection of nearby abstract actions that have not yet been visited. This ASM improved the exploration effectiveness compared to random. Nevertheless, for large and complex SUTs, the total number of actions to be executed for completing the state model inference cannot be reduced.

A distributed architecture can distribute the execution of these actions amongst multiple TESTAR instances to reduce the required time. A centralized state model facilitates the sharing of SUT knowledge generated by the distributed instances. Then, multiple TESTAR instances utilize this shared model to coordinate and select the actions to execute while contributing to enriching the state model collaboratively.

Distributed architectures based on SUT connections

The SUT, for which the model is inferred, can be deployed to manage independent connections or to manage multiple connections with shared data.

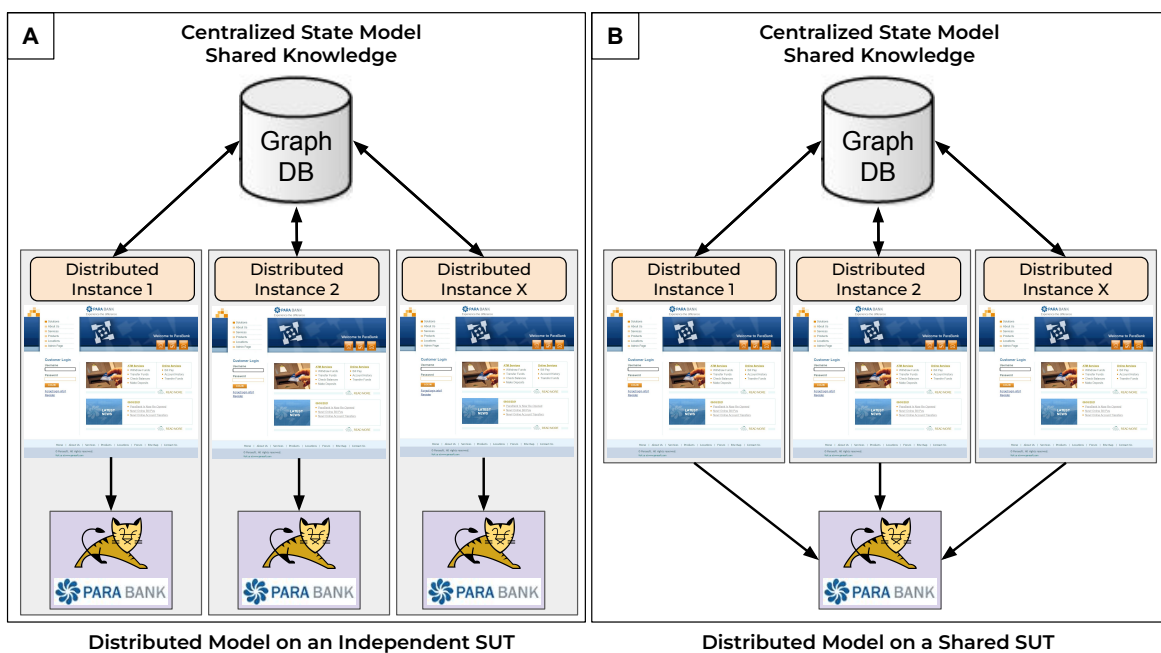


Fig. 4.1 Distributed state model inference architectures

Figure 4.1.A shows an architecture that we call Distributed Model on an Independent SUT (DMIS). This architecture could mitigate concurrent events, but we might find a lack of SUT information. For example, one distributed instance creates a user in an independent SUT instance. This user creation adds new functionality and new widgets in some states of the SUT. However, this new user only exists in that independent SUT, other SUTs have no users created. Then, because all distributed testing instances share the state model knowledge, they may think that new widgets are accessible if they navigate to some specific state. This may cause non-determinism in the shared model and the need to execute the same trace of actions in all the independent SUTs to have the same states with the same data in all the distributed instances.

Figure 4.1.B shows an architecture that we call Distributed Model on a Shared SUT (DMSS). In this case, any distributed instance can alter the data being read by other instances. Therefore, in DMSS, concurrent events might affect the GUI information used to infer the model and provoke additional non-determinism challenges. However, in this case, the distributed instances would discover the new state with the common data. In this research, we focus on using this architecture because it represents most of today's web systems.

Containerization of TESTAR

A distributed architecture requires a separate environment for each distributed instance. For web testing, both the TESTAR tool and the web GUI environment can be effectively containerized. By utilizing the Docker image that includes a standalone version of Selenium ChromeDriver [252], enabling the *X virtual framebuffer* (Xvfb) [301] server to simulate GUI operations, and installing the OpenJDK software, it is possible to execute the TESTAR tool and interact with the web SUT within a Docker instance. That is why we decided to first use Docker for distributed state model inference for web applications and, in the upcoming research, to continue with mobile and desktop applications.

4.1.1 Shared knowledge ASM

When running only one TESTAR instance, the state model algorithm maintains a previously identified shortest path of actions in the model that leads to a previously prioritized unvisited action (i.e., the *targetAction*). However, this approach will not work in a distributed setting because when the tool prioritizes the *targetAction* and selects the actions to get there, another shorter path might become available through states discovered by other instances. Moreover, we need to prevent different instances from prioritizing the same *targetAction*. In a distributed architecture, all instances have to be aware of what other instances are discovering. The instances will do this by reading and writing a shared knowledge repository. To enable this, we developed the first version of a new ASM called *shared knowledge ASM*.

The shared knowledge ASM is used independently by each instance to re-calculate the shortest path to a targeted unvisited action every time a new action is selected. This means that each instance does not maintain a path of actions to reach the state that contains the *targetAction*, but only the *targetAction* that exists in the shared model. Moreover, because there is no centralized component to dispatch the actions to coordinate all the instances, each independent instance has to mark the selected *targetAction* in the shared model to make sure that other instances do not pick the same *targetAction*.

Algorithm **Shared_Knowledge_ASM** presents the shared knowledge ASM for one TESTAR instance. If the TESTAR instance has no *targetAction* (line 1), the algorithm tries to find a new unvisited action closest to the current state s , and marks one of them as *targetAction* in the shared model to prevent other instances from selecting it (line 2). Suppose two different instances try to mark the same *targetAction* at a specific instant of time in the shared model. In that case, OrientDB manages the concurrent events. One of the instances receives an indication to wait a short time before trying to find another unvisited action closest to the current state s and mark them as *targetAction*.

At this point, the TESTAR instance already has marked a *targetAction*. To find the next action to select, the algorithm checks whether the current state s contains the *targetAction* action (line 4). If it does, the algorithm only needs to select the *targetAction* next (line 5) to reach its goal and reset the *targetAction* (line 6), such that the next iteration will search for a new *targetAction* (line 1). If the current state s does not contain the *targetAction* action (line 7), the algorithm calculates the actions that follow the shortest path from s to the target state that contains the *targetAction* (line 8). Because there may be several actions that lead to the shortest path, the algorithm selects one of them randomly (line 9). Finally, the algorithm returns the selected action (line 11).

Algorithm Shared_Knowledge_ASM

Require: s	▷ The state the SUT is currently in
Require: $StateModel$	▷ The state model that is being inferred
Require: $targetAction$	▷ UnvisitedAction marked as targetAction
1: if $targetAction == null$ then	▷ No UnvisitedAction has been marked yet
2: $targetAction \leftarrow popClosestUnvisitedAction(s, StateModel)$	
3: end if	▷ At this point, an UnvisitedAction is marked
4: if $s.contains(targetAction)$ then	▷ If destination state reached
5: $a \leftarrow targetAction$	▷ Select UnvisitedAction marked
6: $targetAction \leftarrow null$	▷ And reset the mark
7: else	▷ Else, move one step closer to the destination state
8: $shortestActions \leftarrow shortestPath(s, targetAction, StateModel)$	
9: $a \leftarrow selectRandom(shortestActions)$	▷ Select one step action
10: end if	
11: return a	▷ Execute selected action

Figure 4.2 shows an example of how multiple distributed instances share knowledge of the state model. The nodes are the discovered states, the transitions between the nodes are the executed actions, and the black hole marks all the actions that were detected but remain unvisited. While *Instance 1* remains currently alone in a state with an unvisited action to execute, *Instance 2* and *Instance 3* are currently both in the same state, and they need to

know what the other instance is going to do. In this case, *Instance 3* has marked first the unvisited action as *targetAction*, then *Instance 2* does not detect that action as unvisited and has decided to mark as *targetAction* an unvisited action that exists one state away.

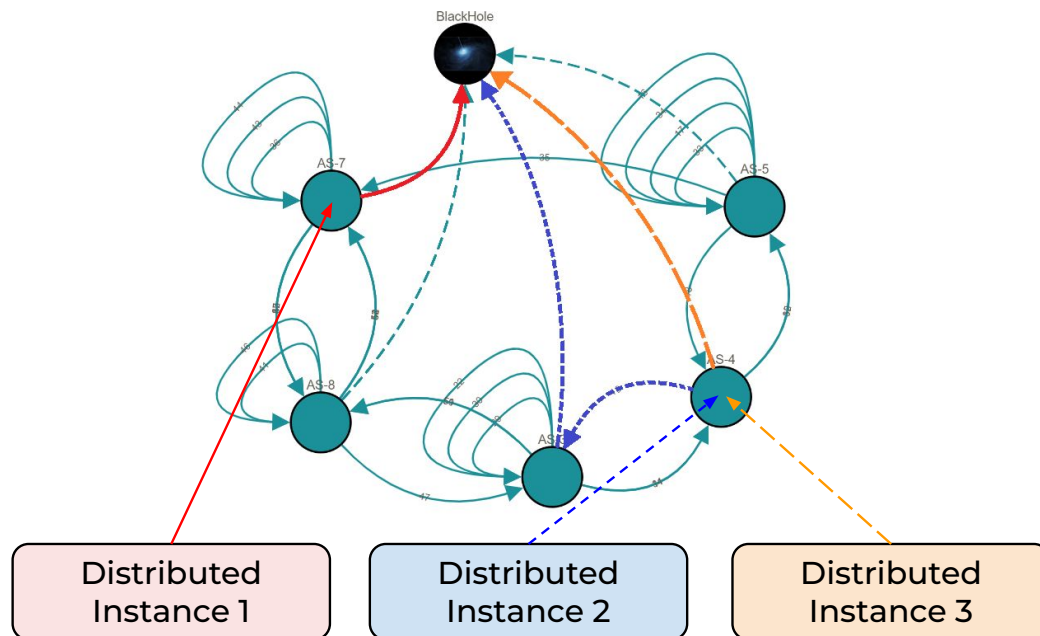


Fig. 4.2 Shared state model knowledge ASM used by multiple distributed TESTAR instances

Table 4.1 Preliminary results of Shared_Knowledge_ASM

instances	runs	states	transitions	avg time	factor
1	10	11	170	1070s	0 %
2	10	11	170	652s	39 %
3	10	11	170	545s	16,5 %
4	10	11	170	530s	2,7 %
5	10	11	170	517s	2,4 %
6	10	11	170	530s	-2,4 %

A preliminary experiment was executed to validate the use of the shared knowledge ASM by multiple Docker instances. To do that, we configured a small web application on which it is possible to infer a complete and deterministic GUI state model. Table 4.1 shows the time it took for the different number of instances to infer the state model. We executed groups from 1 to 6 TESTAR docker instances for 10 runs to obtain the results. A complete run inferred

a state model that consists of 11 states and 170 transitions. The average time indicates the time it has taken for each group of instances to infer the whole model. And the factor is the percentage time reduced between the group of instances and the previous one.

We can see that more distributed instances reduce the time needed to infer the model up to a certain limit. From 1 to 2 and 2 to 3 instances, the inference time is reduced by 39% and 16,5%, respectively. However, from 3 to 5 instances, there is only a small degree of improvement in the speed of the model inference. Moreover, changing from 5 to 6 instances increases the time for this small web application. The instances that do not find an available unvisited action will wait a short time for other instances to enrich the model before trying again. Therefore, while this effect may vary based on the size and complexity of the SUT, we assume that using many instances for small applications will increase the waiting time instead of inferring the model faster.

4.1.2 Non-Deterministic Shared knowledge ASM

The existing previous shared knowledge algorithm worked as expected when exploring small SUTs on which it is possible to infer a deterministic model. However, when applied to real and complex systems, the algorithm was negatively affected by the previously mentioned abstraction challenges: dynamism and non-determinism.

Dealing with dynamism challenges seems feasible by designing abstraction sub-strategies that indicate to TESTAR which widgets or properties it needs to use in different GUI states. However, dealing with non-determinism challenges is not always possible. Sometimes, the required information to differentiate a non-deterministic action is not available through the GUI. Furthermore, the use of the distributed approach with a DMSS architecture increases the issues that can arise related to non-determinism, since the actions executed by other instances can alter the information of the GUI. This challenge cannot be solved only by designing abstraction sub-strategies. In addition, a new algorithm is required to detect and deal with non-deterministic actions, to allow navigation without getting stuck in loops, and to infer a state model covering the reachable parts of the GUI.

To illustrate the challenge of inferring a non-deterministic model, and our solution to deal with it in the new algorithm, we refer to Figure 4.3. Imagine that, at a certain instant of time, during the inference process, a distributed instance I_x discovers that executing action $S1toS2$ in $S1$ transits to state $S2(A)$. For example, an action to transfer a certain amount of money between bank accounts. Now, suppose that in Figure 4.3.A, another instance, I_y , is in state $S1$ and uses the shared knowledge algorithm `Shared_Knowledge_ASM` to mark the *Unvisited Action* of $S2(A)$ as the *targetAction* to execute. However, in Figure 4.3.B, after executing the action $S1toS2$ with the intention to move to state $S2(A)$, I_y realizes it is not in the expected

state. Instead of state $S2(A)$, it is in a new state $S2(B)$ with a new set of *Unvisited Actions* to execute. This could happen, for example, when the transferring account runs out of money. Using algorithm `Shared_Knowledge_ASM`, instance I_y still has the *Unvisited Action* marked in state $S2(A)$, but it will not be able to reach it, at least momentarily. This non-deterministic situation may cause a loop in the inference process.

To avoid such loops, we propose a solution that adds additional functionality to the TESTAR flow. After a transition is saved in the state model, it is checked whether the SUT reached the expected state. When this is not the case, action $S1toS2$ is found to be non-deterministic, and hence all $S1toS2$ actions are changed to point to a *Non-Deterministic Behavior* node (see Figure 4.3.C), meaning they cannot be used for navigating to states $S2(A)$ or $S2(B)$. The algorithm then continues to prioritize first those *Unvisited Actions* to which distributed instances can navigate in the model. When we finished executing all reachable *Unvisited Actions*, the non-deterministic actions are executed again to restore the model transitions. This is illustrated in Figure 4.3.D, where the non-deterministic action $S1toS2$ that discovered the transition to state $S2(B)$ (and that was previously removed in Figure 4.3.C) is restored, and the transition pointer to the *Non-Deterministic Behavior* is deleted. After that, we can detect new *Unvisited Actions* in state $S2(B)$ and continue the exploration.

In the new Algorithm `Non_Deterministic_Shared_Knowledge_ASM`, a new variable, *expectedState*, is added to the operational flow of TESTAR. This variable is updated when the tool selects an action to execute. After that, it is used in the new Method `Verify_Action_Transition_Determinism` when executing the selected action to save the new transition in the model. This new variable enables the detection of non-determinism in the path that leads to the *targetAction*.

Algorithm `Non_Deterministic_Shared_Knowledge_ASM` first prioritises unvisited actions as *targetAction* (lines 1 to 3). If no unvisited actions are available, the instance has no *targetAction* (line 4). Then, the algorithm tries to find a new non-deterministic action closest to the current state s . The non-deterministic action is marked as *targetAction* in the state model (line 5) to prevent other instances from selecting it. To find the next action to select, the algorithm checks whether the current state s contains the *targetAction* action (line 7). If it does, the algorithm only needs to select the *targetAction* to reach the target (line 8), reset *targetAction*, such that the next iteration will search for a new *targetAction* (line 9), and reset the *expectedState* because we reached the *targetAction* without non-determinism issues (line 10). If the current state s does not contain the *targetAction* action (line 11), the algorithm calculates the actions that follow the shortest path from s to the target state that contains the *targetAction* (line 12). Because there may be several actions that lead to the shortest path, the algorithm selects one of them randomly (line 13). Here, we update the *expectedState*

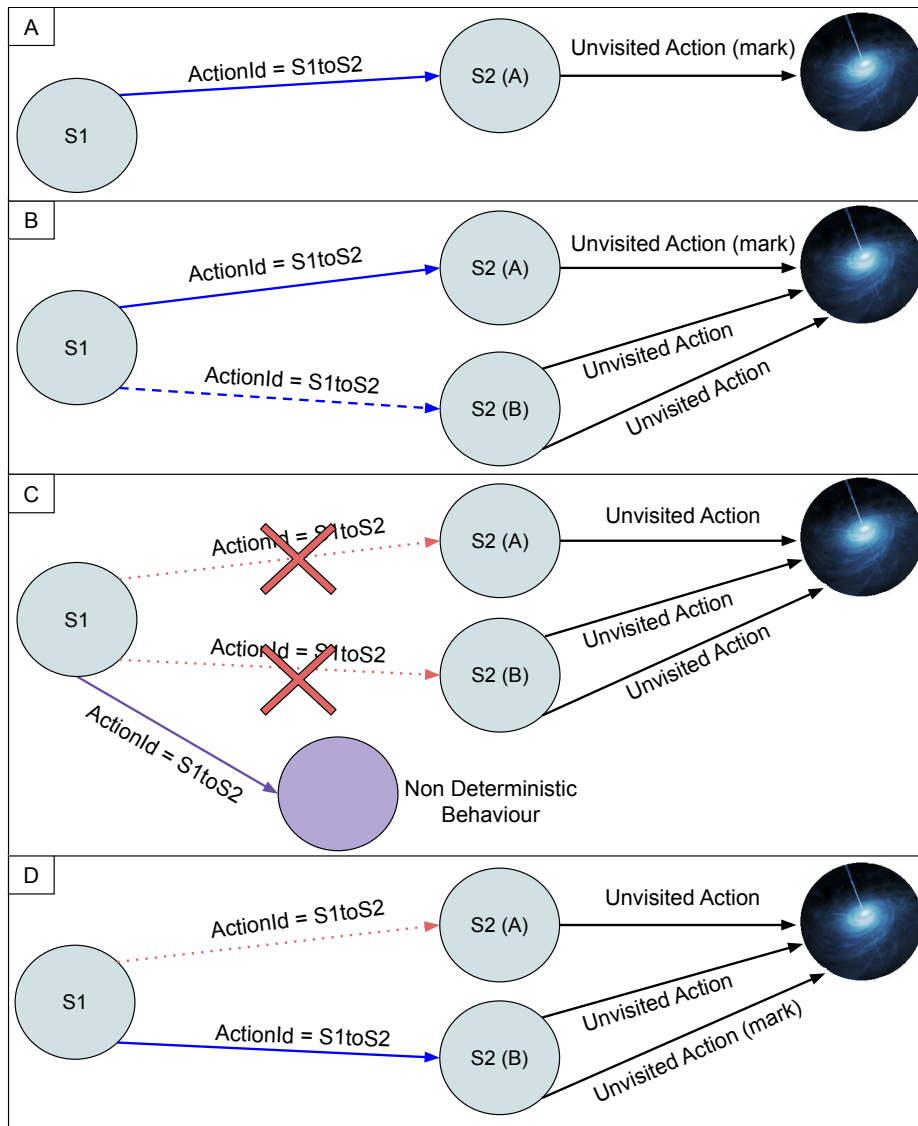


Fig. 4.3 Dealing with non-deterministic models

variable with the state we should navigate if the transition is deterministic (line 14). Finally, the algorithm returns the selected action (line 16).

Following the TESTAR flow, this selected action is executed, which results in a state transition that is added to the model. This is the moment where the *expectedState* variable is used to detect non-determinism. This is shown in Method `Verify_Action_Transition_Determinism` in case the *expectedState* is defined.

If the state to which the model has transitioned after executing the selected action is not the *expectedState*, the action is non-deterministic because it leads to two different states (line 1). In this case, the action is marked as non-deterministic in the model (line 2). Subsequently,

Algorithm Non_Deterministic_Shared_Knowledge_ASM

Require: s ▷ The state the SUT is currently in
Require: $StateModel$ ▷ The state model that is being inferred
Require: $targetAction$ ▷ UnvisitedAction or NonDeterministicAction marked as
targetAction
Require: $expectedState$ ▷ Expected next step state to navigate to deterministically

- 1: **if** $targetAction == null$ **then** ▷ If no UnvisitedAction or NonDeterministicAction
marked
- 2: $targetAction \leftarrow popClosestUnvisitedAction(s, StateModel)$
- 3: **end if** ▷ At this point, an UnvisitedAction is marked if available
- 4: **if** $targetAction == null$ **then** ▷ If no UnvisitedAction available in previous condition
- 5: $targetAction \leftarrow popClosestNonDeterministicAction(s, StateModel)$
- 6: **end if** ▷ At this point, a NonDeterministicAction is marked if available
- 7: **if** $s.contains(targetAction)$ **then** ▷ If destination state reached
- 8: $a \leftarrow targetAction$ ▷ Select the UnvisitedAction or NonDeterministicAction marked
- 9: $targetAction \leftarrow null$ ▷ Reset the target action mark
- 10: $expectedState \leftarrow null$ ▷ Reset the determinism state check
- 11: **else** ▷ Else, move one step closer to the destination state
- 12: $shortestActions \leftarrow shortestPath(s, targetAction, StateModel)$
- 13: $a \leftarrow selectRandom(shortestActions)$ ▷ Select one-step action
- 14: $expectedState \leftarrow nextDeterministicState(a)$ ▷ Save the next step state we should
navigate to deterministically
- 15: **end if**
- 16: **return** a ▷ Return the selected action

Method Verify_Action_Transition_Determinism

Require: $expectedState \neq null$ ▷ The expected state to navigate deterministically

- 1: **if** $currentState \neq expectedState$ **then**
- 2: $markActionAsNonDeterministic(a)$
- 3: $targetAction \leftarrow null$
- 4: $expectedState \leftarrow null$
- 5: **end if**

the $targetAction$ is reset, so the next iteration will search for a new $targetAction$ (line 3). Finally, the $expectedState$ variable is reset, because the next iteration will follow a new path (line 4).

4.1.3 Empirical Evaluation

The objective of the distributed implementation is to improve the speed of the state model inference process. The aim is to measure whether the distributed approach is more efficient

in terms of the time needed to explore the SUT. The following research question guides this study:

- *Distributed-RQ1*: Does a distributed architecture allow scriptless testing tools to speed up state model inference?

To answer the *Distributed-RQ1*, a controlled experiment is designed by following the guidelines suggested by Wohlin et al. [297]. This study is part of a knowledge repository for secondary studies that we are filling following a methodological framework specifically built to evaluate scriptless testing techniques [278]. Encouraged to disseminate the results, open-source systems were chosen for the experimentation. This experiment consists of a set of trials executing the distributed approach. In each trial, variables related to the time efficiency of the SUT exploration and the state model inference process are measured.

To make statistical analysis possible, we formulate the following null hypothesis based on the research question:

- *Distributed – H₀*: The distributed approach does not reduce the time required to infer a state model.

SUT objects

The SUTs selected for this experiment, suitable for the methodological framework, must be published under an open-source license and should comply with the following:

1. TESTAR is able to detect the GUI of the SUT;
2. The SUTs can be deployed without a manual database configuration, which facilitates the automation and replication of the experiments;
3. The SUTs are Java based. This allows using JaCoCo [127], a well-known and widely used tool in the academic and industry sector, to obtain instruction and branch code coverage metrics for the empirical evaluation.

Based on these requirements, the following SUTs were selected:

- Parabank [211] is an open-source online banking demo application that uses a Java implementation on the server side to process the management of customers, transactions, payments, and other bank services. This demo web application developed by Parasoft company is a representative SUT to test standard Java web services.

- Shopizer [69] is an open-source Java e-commerce web app with two different domains: the administrator pages and the shop for customers. This web application is widely used by companies to offer product sales services. We leave out the admin section in these experiments and focus on the exploration of the shop.

We present the details of the selected SUTs in Table 4.2. The number of internal Java classes, methods, lines of code (LLOC), and instructions and branches indicate that these are not toy projects, i.e. they are representative SUTs of real applications. Moreover, these SUTs expose relevant GUI challenges, such as dynamism and non-determinism, that need to be faced in today's complex systems.

Table 4.2 Details of the selected SUTs

Metric	Parabank	Shopizer
SUT type	Web	Web
Java Classes	153	330
Methods	1127	2356
LLOC	4182	16398
Instructions	18288	74926
Branches	1214	5258

It was necessary to use the *blocking principle* [297] and disable the TESTAR oracles. This is because the usage of TESTAR oracles for fault detection involves stopping, reporting, and restarting the System Under Test (SUT). This feature affects the exploratory time used by the TESTAR instances, reducing the number of states discovered and actions executed. Using the *blocking principle*, we could focus the experiment only on the time required for the model inference process. The usage of oracles to detect failures will be investigated in future experiments. In order to prepare TESTAR for the experiment, we have configured a set of independent variables for each SUT that will be explained next.

Independent variables: time

A set of time variables are used to define the time budget of the experiments and the execution and waiting time of actions. It was necessary to run preliminary executions to determine an appropriate time budget for each SUT. Because Parabank is a web application formed mainly by a series of static GUI forms to use the bank services, 30 minutes was enough to reach a limit on the number of discovered states and decrease the unvisited actions. However, for

Shopizer, it was necessary to increase the time budget until 240 minutes. This is because the web shop application is formed with a dynamic GUI that allows search, add, remove, and order products, creating a high amount of available states and action in the state model.

The time used to execute a GUI action is the same for both SUTs. Nevertheless, the time to wait between actions has been increased for Shopizer because some web states require more time to completely load the data in the GUI of some of the products. Table 4.3 shows the set of time variables used in our experiments for both SUTs.

Table 4.3 Independent time variables

Variable	Parabank	Shopizer
Time budget	30 minutes	240 minutes
Action Duration	1 second	1 second
Wait After Action	1 seconds	2 seconds

Independent variables: derived actions

Regardless of the execution of a unique instance or the distributed approach, using the default set of derived actions that TESTAR realizes automatically is not enough to adequately simulate user interactions and effectively explore complex systems. In each SUT we need to derive smarter actions to maintain the exploration in the main GUI states or to reach some GUI states that require a specific combination of data or actions:

- *Force* actions are used to start and maintain the exploration in the desired state of the SUTs.
- *Filter* actions can be used to maintain the exploration in the main part of the SUT or save time by ignoring less essential features that may increase the exploration space.
- *Custom* actions are created for SUT specific widgets (e.g., top level menus, dropdown menus, shopping carts, etc).
- *Compound* actions are added to the set of derived actions to deal with the filling of forms in one (compound) action.

Table 4.4 shows the derived smarter actions used in our experiments for Parabank and Shopizer.

Table 4.4 Independent variables for smarter action derivation

Parabank	(Force) User login on each sequence (Force) Web history back on WSDL pages (Filter) Logout and admin panel buttons (Filter) Links outside parabank domain (Compound) Form to transfer a valid/invalid amount of money (Compound) Form to request a valid/invalid money loan (Compound) Form to find transactions by id, date or amount (Compound) Form to update customer information (Compound) Form for bill payment (Compound) Form for contact service
Shopizer	(Force) Accept cookies (Force) User login on each sequence (Filter) Logout and send mail buttons (Filter) Change language button (Custom) Actions in top level menus (Custom) Move mouse action for dropdown widgets (Custom) Only add product to cart if shopping cart is empty (Compound) Form to send an email to contact service (Compound) Form to change customer address (Compound) Form to change the account password (Compound) Form to realize the payment of the cart

Sliding actions offer an additional challenge when inferring a state model. These actions change the visible widgets in the GUI that can be interacted with, hence the abstract state. Depending on the SUT, the set of possible sliding positions can increase the exploration space. For this reason, and because it is possible to explore both SUTs without the need to execute sliding actions, we decided not to derive sliding actions.

Independent variables: abstraction strategy

TESTAR's default abstraction mechanism cannot differentiate type actions that type different texts, nor deal with the dynamism issues mentioned previously. Therefore, a new abstraction strategy that distinguishes a main mechanism and a set of sub-strategies, has been implemented in the tool by extending the default mechanism in the Java protocol of the tool.

Table 4.5 shows the main abstraction mechanism to identify the abstract states and actions of Parabank and Shopizer. Abstracting from dynamism can cause the same widget to change its hierarchical index in the widget tree of the abstract state. For this reason, instead of just

using the widget tree hierarchical index for the abstract action, we use the origin widget abstract identifier. Compound actions now have a different abstraction from default actions because the content of the typed text is important when identifying an action transition in the state model.

Table 4.5 Independent variables for the main abstraction mechanism

Parabank	
State	WebId, WebName, WebTextContent
Action	OriginState + OriginWidget + ActionRole
CompoundAction	OriginState + OriginWidget + ActionTypedText
Shopizer	
State	WebId, WebTextContent, WidgetIsOffscreen
Action	OriginState + OriginWidget + ActionRole
CompoundAction	OriginState + OriginWidget + ActionTypedText

Using only the main abstraction mechanism has limitations when applied in systems with dynamism. Each SUT may need the design of a set of abstraction sub-strategies to filter the dynamic widgets and their properties to avoid a state space explosion:

- Widgets that can be dynamically added or removed from the state, such as a dynamic table of elements or products of a shopping cart, are *ignored* when the tool generates the identifier of the abstract state.
- Widgets that contain properties with values that can change dynamically throughout the exploration, such as a dynamic text that displays user-editable information, need to use a *custom* set of specific properties to avoid using these dynamic values.
- Widgets that don't have enough default property values to differentiate them, such as dropdown widgets that don't contain a different web identifier, require new properties to be *added* to the abstraction mechanism.
- Widgets that respond to hover mouse events or states that require a longer time to load all web elements may provoke non-determinism in the model. Shopizer contains additional *events* intended to prevent this non-determinism.

Table 4.6 shows the design of abstraction sub-strategies for Parabank and for Shopizer.

Table 4.6 Independent variables for abstraction sub-strategies

Parabank	(Ignored) Dynamic widgets from accounts table (Custom) Use only WebId for widgets of type select (Custom) Use only WebName for phone number widget (Custom) Use only WebId for new account widget (Custom) Use only WebId for widgets of bill payment state
Shopizer	(Ignored) Price and product widgets in the shopping cart (Ignored) Product and order number widgets in payment forms (Ignored) Address content in customer box info (Custom) Use only WebId in form widgets that dynamically change hint text (Custom) Differentiate empty cart and cart with products (Added) Use parent id for edit bill and shipment widgets (Added) Use previous state id for Apache 404 state (Added) Use parent text content for dropdown widgets (Added) Use hyperlink reference for Home widgets (Event) Force mouse coordinates before each sequence (Event) Wait until loading overlay is complete

Effort time for independent variables

Obtaining the right values for these independent variables requires effort to identify, prepare, and test. Most effort is needed to detect dynamic widgets and design the set of abstraction sub-strategies to implement the main abstraction mechanism for each SUT.

Parabank required approximately 8 hours to prepare the smarter action derivation and 20 hours to determine the best combination of properties for the main abstraction mechanism together with the design of abstraction sub-strategies. More than half of the time was used to test the implementation in the distributed docker architecture and verify the correct identification of states and actions in the model.

Shopizer, on the other hand, required approximately 16 hours to prepare the smarter action derivation and 40 hours to determine the best combination of properties for the main abstraction and the set of abstraction sub-strategies. This is a larger SUT regarding the number of discovered GUI states and actions which required more testing and verification time.

The time invested to learn and design smart actions and abstraction sub-strategies for a SUT may vary according to the user's expertise. It is important to note that the knowledge acquired helps to reduce the learning time in future configurations of similar SUTs. Table 4.7 contains an approximation of the times required by a TESTAR member to prepare the independent variables for Parabank and Shopizer.

Table 4.7 Effort time to design the independent variables

Time required for variable	Parabank	Shopizer
Smarter action derivation	8 hours	16 hours
Abstraction Strategies	20 hours	40 hours

Dependent variables

To answer Distributed-RQ1, we need to measure whether, given a time budget, the distributed approach improves the speed of exploring and inferring a model. Thus, to compare the speed of the distributed approach, the following metrics are measured:

1. *Code coverage* over time to determine which parts of the code have been executed during the testing.
2. *GUI coverage* over time to determine how many different GUI states and actions have been found in the SUT. For a fair comparison, we will use the same abstraction strategy for all the trials of the same SUT.

Since a distributed implementation requires more computational resources, the following metric is also measured:

3. *Consumption* metrics to measure the CPU and memory usage for each TESTAR Docker instance, the SUT, and the central graph database.

Design of the experiment

We compare the distributed approach with Parabank and Shopizer by running groups from 1 to 6 Docker testing instances, i.e., the first group with a single Docker testing instance, the second group with two distributed Docker testing instances, and so on, up to six distributed Docker testing instances. Each group of Dockers runs on an Ubuntu machine with 8 CPU cores and 32GB RAM. All Docker containers are deployed with 512M shared memory size. To deal with the randomness and to be able to obtain valid conclusions about the possible rejection of the null hypotheses, we repeat the execution of the experiment 30 times. The SUT will start in the same initial status in each new execution. To do this, a bash controller takes care of:

1. Restart the system to remove possible residual processes.
2. Clean and copy Apache Tomcat with the web SUT, the graph database, and the TESTAR files; and prune the Docker containers of the previous executions.

3. Launch a custom Java application and bash instructions to extract code coverage, state model GUI coverage, and consumption metrics every 5 seconds.
4. Build and deploy Apache Tomcat with the web SUT, the graph database, and a new number of Docker containers with TESTAR to start the testing process.

TESTAR logs and HTML output results, together with the coverage, state model, and performance metrics, are stored in a centralized data server at the end of each execution. Figure 4.4 shows the overall design of the experiment.

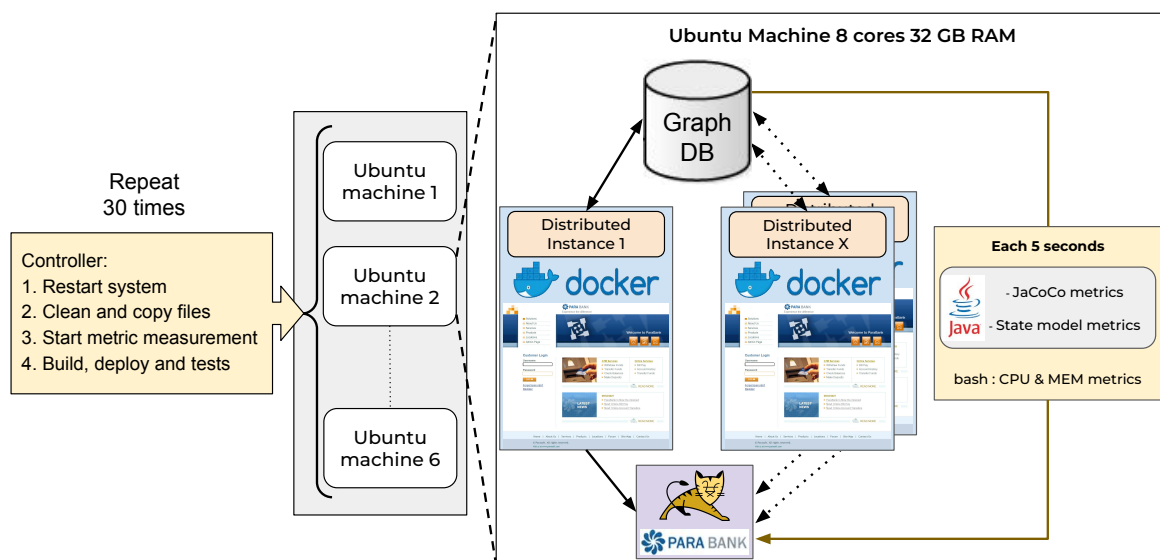


Fig. 4.4 Architecture for the execution of distributed experiments

4.1.4 Results

This section presents the results of the distributed approach for Parabank and Shopizer. Instruction code coverage is presented in curves comparing 1 with 6 groups of Dockers, and comparing the average code coverage of all Docker instances in a growth curve. We only show instruction coverage because branch coverage is visually similar. The graph curves for the six groups of Dockers are available online¹. However, the subsequent statistical analysis is performed for both instructions and branch coverage. GUI coverage is presented in curves comparing 1 with 6 groups of Dockers. We use the Kruskal Wallis test for the instruction and branch coverage to check whether there is a significant difference between the six different

¹<https://doi.org/10.5281/zenodo.7607233>

Docker groups [93]. Then, we use Wilcoxon and Cliff's delta tests to check which groups of Docker pairs differ significantly.

Parabank analysis

For a single Docker instance, the metrics obtained with JaCoCo indicates that the code coverage increases throughout the first 30 minutes of exploration (see Figure 4.5). While, for six instances, the total code coverage is reached approximately in 10 minutes. For GUI coverage, we evidently observe a similar trend with the metrics extracted from the TESTAR state model (see Figure 4.6). As new GUI abstract states are discovered in the TESTAR state model that represents the SUT, the code covered obtained with JaCoCo increases correspondingly.

Concerning the number of unvisited abstract actions for the GUI coverage (see Figure 4.6), new unvisited actions are found during the first 20 minutes for a single Docker instance. Then, in the last 10 minutes, the number of unvisited actions starts decreasing. For six Docker instances, the peak of unvisited actions is reached in the first 5 minutes. After that, it decreases during 10 minutes, reaching a limit that does not go to zero. The latter is because some unvisited actions are unreachable due to non-determinism in the inferred models.

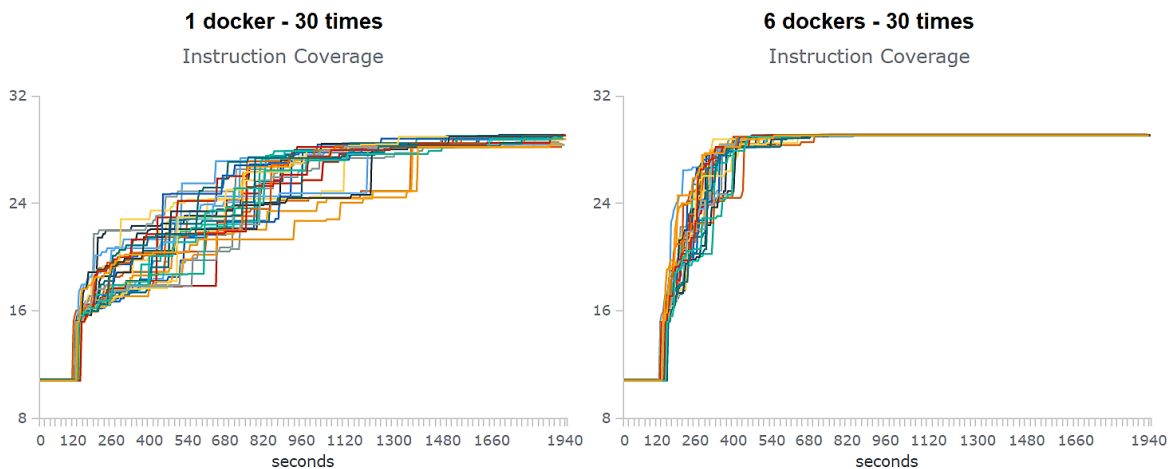


Fig. 4.5 Parabank code coverage comparison for 1 and 6 groups of Docker. Each line represents one of the thirty repeated executions.

Figure 4.7 represents the average code coverage of all Docker instances for Parabank in the first 700 seconds of execution. It is easy to see a growth difference between all the Docker groups except for 3 and 4 groups of Docker.

We perform statistical analysis at 360, 480, and 600 seconds to obtain evidence about the significant difference in the distributed approach. This allows us to analyse how the

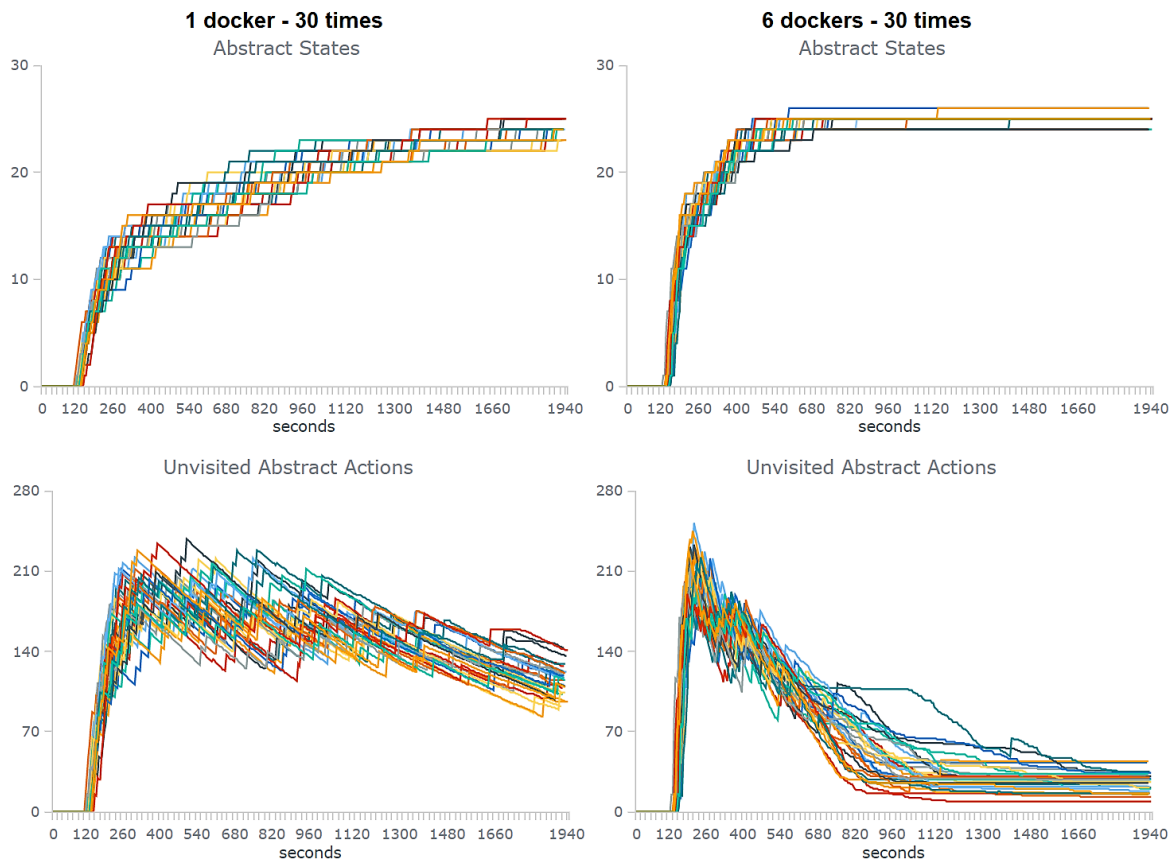


Fig. 4.6 Parabank state model comparison for 1 and 6 groups of Dockers. Each line represents one of the thirty repeated executions.

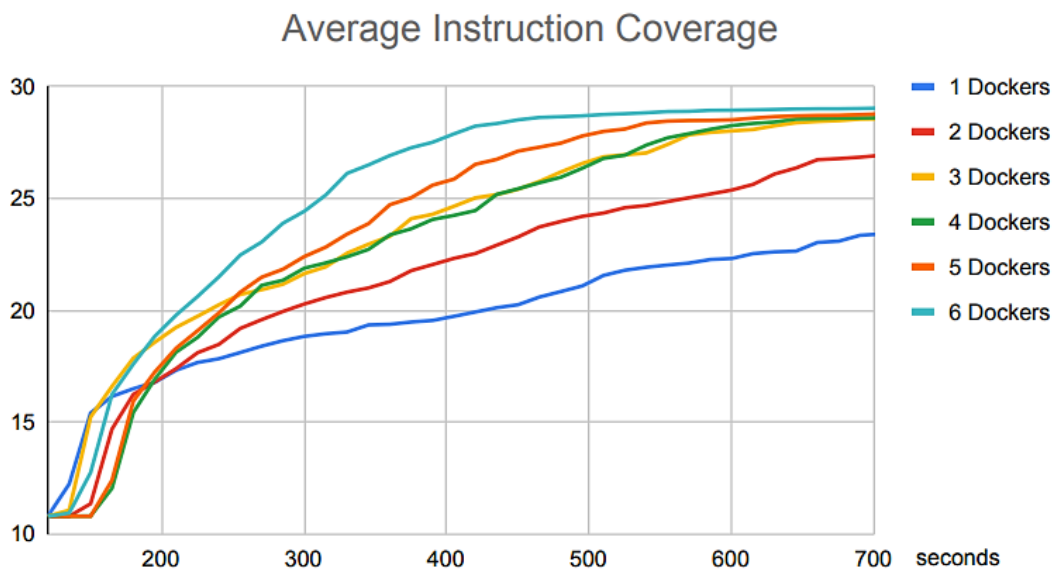


Fig. 4.7 Parabank average code coverage growth for each group of Dockers.

distributed approach evolves from the moment the distributed instances have started exploring until the moment the groups with the most Dockers have reached the coverage limit. The Kruskal Wallis test indicates a significant difference between all distributed Docker groups in these three instants of time (see Table 4.8).

Table 4.8 Parabank Kruskal Wallis results

Parabank	Instruction Coverage		Branch Coverage	
	H-value	P-value	H-value	P-value
360 seconds	115.11	3.39e-23	113.33	8.09e-23
480 seconds	130.17	2.19e-26	130.3	2.05e-26
600 seconds	147.18	5.32e-30	148.07	3.44e-30

At 360 seconds (see Table 4.9), Wilcoxon indicates no significant difference only for 3 & 4 groups of Dockers and a significant difference for the other groups of Dockers. Cliff’s delta points negligent difference for 3 & 4 groups of Dockers, medium difference for 1 & 2, 3 & 5, and 4 & 5 groups of Dockers, and large difference for the other groups.

Table 4.9 Parabank significant difference at 360 seconds

Wilcoxon: p-value (p), Cliff’s δ : small(S), medium(M) and large (L) difference

Cliff’s δ	Instruction Coverage						Branch coverage					
	1 Dockers	2 Dockers	3 Dockers	4 Dockers	5 Dockers	6 Dockers	1 Dockers	2 Dockers	3 Dockers	4 Dockers	5 Dockers	6 Dockers
1 Dockers		L -0.507	L -0.895	L -0.895	L -0.99	L -1.0		M -0.436	L -0.898	L -0.861	L -0.979	L -0.999
2 Dockers	p 0.004165		L -0.543	L -0.555	L -0.781	L -0.951	p 0.007078		L -0.592	L -0.515	L -0.758	L -0.936
3 Dockers	p 0.000011	p 0.001754		N 0.03	M -0.413	L -0.858	p 0.000013	p 0.002707		S 0.162	M -0.441	L -0.847
4 Dockers	p 0.000012	p 0.000334	p 0.926255		M -0.457	L -0.83	p 0.000011	p 0.000777	p 0.462133		L -0.496	L -0.847
5 Dockers	p 0.000010	p 0.000027	p 0.004165	p 0.008283		L -0.668	p 0.000010	p 0.000032	p 0.013704	p 0.010542		L -0.652
6 Dockers	p 0.000010	p 0.000010	p 0.000011	p 0.000053	p 0.000334		p 0.000010	p 0.000010	p 0.000012	p 0.000034	p 0.000648	

At 480 seconds (see Table 4.10), Wilcoxon continues indicating no significant difference for 3 & 4 groups of Dockers and a significant difference for the other groups of Dockers. Whereas Cliff’s delta has changed and only indicates negligent difference for 3 & 4 groups of Dockers, the other Docker groups have large differences.

Table 4.10 Parabank significant difference at 480 seconds

Wilcoxon: p-value (p), Cliff’s δ : small(S), medium(M) and large (L) difference

Cliff’s δ	Instruction Coverage						Branch coverage					
	1 Dockers	2 Dockers	3 Dockers	4 Dockers	5 Dockers	6 Dockers	1 Dockers	2 Dockers	3 Dockers	4 Dockers	5 Dockers	6 Dockers
1 Dockers		L -0.672	L -0.971	L -0.903	L -1.0	L -1.0		L -0.662	L -0.967	L -0.91	L -0.993	L -1.0
2 Dockers	p 0.000047		L -0.541	L -0.531	L -0.812	L -1.0	p 0.000108		L -0.6	L -0.554	L -0.828	L -1.0
3 Dockers	p 0.000005	p 0.002136		N 0.003	L -0.554	L -0.98	p 0.000006	p 0.003292		N 0.011	L -0.531	L -0.973
4 Dockers	p 0.000008	p 0.002982	p 0.853135		L -0.508	L -0.95	p 0.000006	p 0.002482	p 0.696972		L -0.486	L -0.946
5 Dockers	p 0.000005	p 0.000016	p 0.000337	p 0.003620		L -0.791	p 0.000006	p 0.000020	p 0.000363	p 0.005343		L -0.797
6 Dockers	p 0.000005	p 0.000005	p 0.000005	p 0.000005	p 0.000041		p 0.000006	p 0.000006	p 0.000006	p 0.000006	p 0.000026	

Finally, at 600 seconds (see Table 4.11), Wilcoxon indicates that all Docker groups have significant difference, including 3 & 4 groups of Dockers. And Cliff’s delta points that 3 & 4

and 4 & 5 Docker groups have small significant difference for instruction coverage, medium difference for 4 & 5 groups of Dockers regarding branch coverage, while the other groups of Dockers have large difference.

Table 4.11 Parabank significant difference at 600 seconds
Wilcoxon: p-value (p), Cliff's δ : small(S), medium(M) and large (L) difference

Cliff's δ Wilcoxon	Instruction Coverage						Branch coverage					
	1 Dockers	2 Dockers	3 Dockers	4 Dockers	5 Dockers	6 Dockers	1 Dockers	2 Dockers	3 Dockers	4 Dockers	5 Dockers	6 Dockers
1 Dockers		L -0,732	L -0,988	L -0,997	L -1,0	L -1,0		L -0,735	L -0,982	L -0,993	L -1,0	L -1,0
2 Dockers	p 0,00001		L -0,913	L -0,962	L -1,0	L -1,0	p 0,000012		L -0,912	L -0,962	L -1,0	L -1,0
3 Dockers	p 0,000005	p 0,00003		S -0,283	L -0,496	L -0,933	p 0,000005	p 0,000042		S -0,275	L -0,534	L -0,941
4 Dockers	p 0,000005	p 0,00001	p 0,034456		S -0,271	L -0,924	p 0,000005	p 0,000012	p 0,018889		M -0,33	L -0,943
5 Dockers	p 0,000005	p 0,000005	p 0,001463	p 0,016825		L -0,823	p 0,000005	p 0,000005	p 0,001026	p 0,010314		L -0,836
6 Dockers	p 0,000005	p 0,000005	p 0,000005	p 0,000006	p 0,000024		p 0,000005	p 0,000005	p 0,000006	p 0,000008	p 0,000013	

Related to Parabank computation resource consumption, Table 4.12 presents the CPU and memory consumption average of all groups of Dockers. CPU and memory consumption are increased around 6–7% and 5–6% respectively, for each new distributed Docker instance executed.

Table 4.12 Parabank Consumption Average

Dockers	1	2	3	4	5	6
CPU	16,57%	23,93%	29,86%	36,76%	41,49%	47,49%
MEM	23,85%	29,45%	34,90%	40,46%	45,38%	50,27%

Shopizer analysis

Regarding Shopizer code coverage (see Figure 4.8), a single Docker instance needs 2 hours or more to reach the maximum coverage. While, with six Dockers, except for a couple of outliers, the maximum coverage is reached in approximately one hour of execution.

The maximum instruction coverage reached by the TESTAR instances in all groups of dockers was approximately 18.3%, except for one specific instance in the group of 5 dockers that reached 23.5% of coverage. We consider that a set of random actions led to purchasing or modifying a product in the store, increasing the code coverage.

For GUI coverage (see Figure 4.9), the group of 6 Dockers needs approximately 1 hour to discover the peak of abstract states. Then, we can observe a decrease in the unvisited abstract actions curve. Otherwise, the group of 1 Docker keeps discovering new states throughout the 4 hours of execution. There is an execution time in which there is no noticeable decrease in the number of unvisited abstract actions.

With Shopizer, after 1 hour of execution with a group of 6 Dockers, the number of unvisited actions slowly drops until 3 hours of execution, when it seems to reach a limit. As with Parabank, we assume this is due to unreachable unvisited actions because of non-determinism in the model.

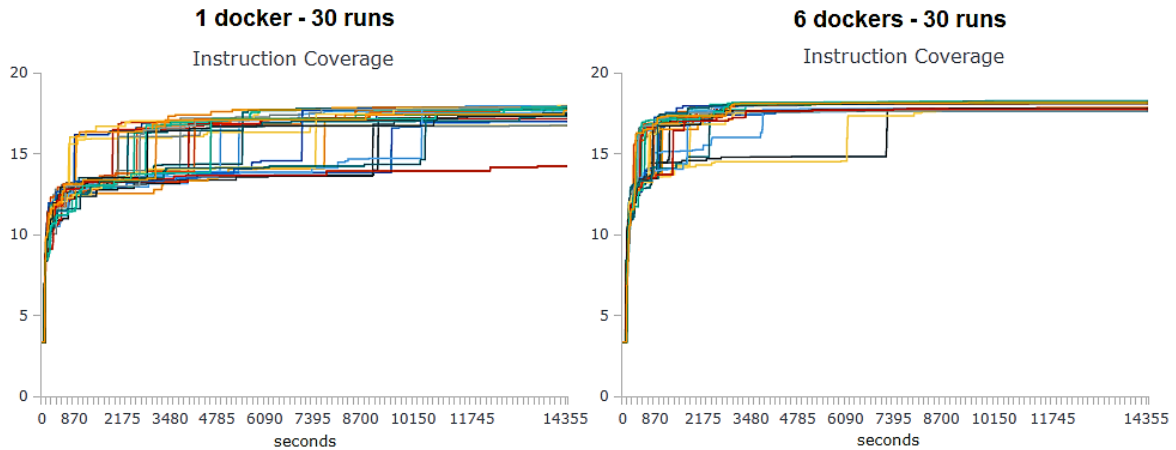


Fig. 4.8 Shopizer code coverage comparison for 1 and 6 groups of Dockers. Each line represents one of the thirty repeated executions.

Figure 4.10 represents the average code coverage of all Docker instances for Shopizer in the first hour of execution. We can see a growth difference between all groups of Dockers. However, the groups of 3, 4, 5, and 6 Dockers seem to differ only slightly.

We perform statistical analysis at 500, 1750 and 3000 seconds to obtain evidence about the significant difference in the distributed approach. From the moment the distributed instances have started exploring to the moment the groups with the most Dockers have reached the coverage limit. The Kruskal Wallis test indicates a significant difference between all distributed Docker groups in these three instants of time (see Table 4.13).

Table 4.13 Shopizer Kruskal Wallis results

Shopizer	Instruction Coverage		Branch Coverage	
	H-value	P-value	H-value	P-value
500 seconds	78.74	1.54e-15	74.06	1.46e-14
1750 seconds	82.53	2.48e-16	88.79	1.21e-17
3000 seconds	89.72	7.71e-18	105.74	3.25e-21

At 500 seconds (see Table 4.14), Wilcoxon indicates no significant difference for 2 & 3, 2 & 4, 3 & 4, 3 & 5, and 5 & 6 groups of Dockers, including 3 & 6 groups of Dockers only

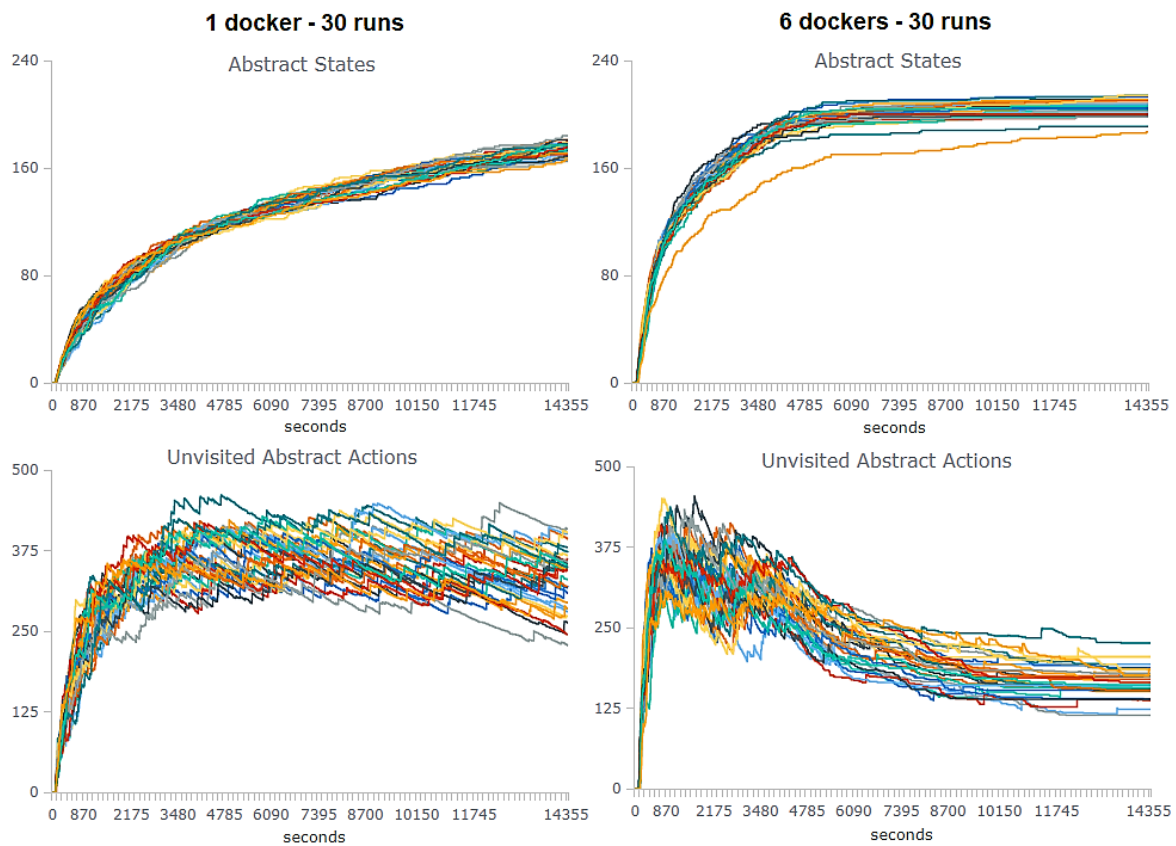


Fig. 4.9 Shopizer state model comparison for 1 and 6 groups of Dockers. Each line represents one of the thirty repeated executions.

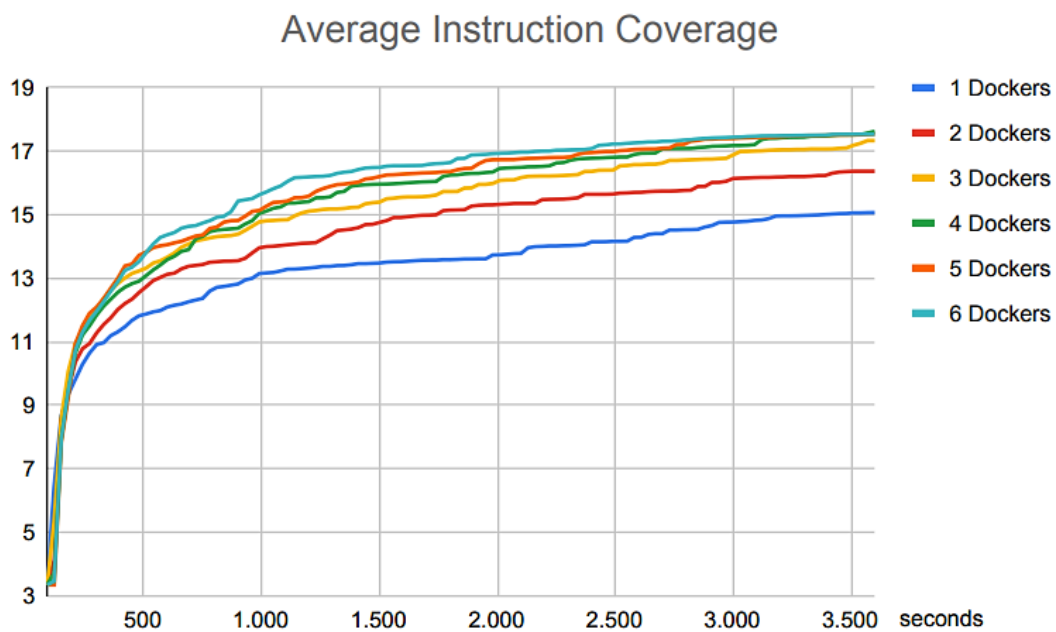


Fig. 4.10 Shopizer average code coverage growth for each group of Dockers.

regarding branch coverage. Then, a significant difference for the other groups of Dockers. And Cliff’s delta points negligent difference for 3 & 4 and 5 & 6 groups of Dockers. Medium difference for 2 & 3, 2 & 4, 3 & 5, 3 & 6, 4 & 5, and 4 & 6 groups of Dockers, including 1 & 2 groups of Dockers only regarding branch coverage. Then a large difference for the other groups is shown.

Table 4.14 Shopizer significant difference at 500 seconds
Wilcoxon: p-value (p), Cliff’s δ : small(S), medium(M) and large (L) difference

Cliff’s δ Wilcoxon	Instruction Coverage						Branch coverage					
	1 Dockers	2 Dockers	3 Dockers	4 Dockers	5 Dockers	6 Dockers	1 Dockers	2 Dockers	3 Dockers	4 Dockers	5 Dockers	6 Dockers
1 Dockers		L -0,568	L -0,796	L -0,821	L -0,962	L -0,964		M -0,464	L -0,725	L -0,76	L -0,938	L -0,91
2 Dockers	p 0,003412		M -0,395	M -0,413	L -0,703	L -0,688	p 0,017407		M -0,434	M -0,435	L -0,708	L -0,696
3 Dockers	p 0,000042	p 0,085892		N -0,025	M -0,37	M -0,37	p 0,000199	p 0,086785		N 0,012	M -0,384	M -0,346
4 Dockers	p 0,000036	p 0,096516	p 0,869294		M -0,395	M -0,362	p 0,000049	p 0,080592	p 0,638817		M -0,457	M -0,397
5 Dockers	p 0,000019	p 0,003217	p 0,077814	p 0,010939		N 0,073	p 0,000029	p 0,003453	p 0,080592	p 0,005351		N 0,087
6 Dockers	p 0,000019	p 0,001711	p 0,047378	p 0,015446	p 0,869294		p 0,000032	p 0,001469	p 0,080592	p 0,016355	p 0,643508	

At 1750 seconds (see Table 4.15), Wilcoxon and Cliff’s delta differ moderately from instruction and branch coverage. For instruction coverage, Wilcoxon specify no significant difference for 2 & 3, 3 & 4, 3 & 5, 4 & 5, 4 & 6, and 5 & 6 groups of Dockers and a significant difference for the other groups of Dockers. Whereas Cliff’s delta points negligent difference for 4 & 5 and 5 & 6 groups of Dockers, small difference for 3 & 4 and 4 & 6, medium difference for 2 & 3 and 3 & 5 groups of Dockers, and large difference for the other groups.

While, for branch coverage, Wilcoxon indicates no significant difference only for 3 & 4, 4 & 5, and 5 & 6 groups of Dockers and a significant difference for the other groups of Dockers. And Cliff’s delta points negligent difference for 5 & 6 groups of Dockers, small difference for 3 & 4 and 4 & 5, medium difference for 2 & 3, 3 & 5, and 4 & 6 groups of Dockers, and large difference for the other groups.

Table 4.15 Shopizer significant difference at 1750 seconds
Wilcoxon: p-value (p), Cliff’s δ : small(S), medium(M) and large (L) difference

Cliff’s δ Wilcoxon	Instruction Coverage						Branch coverage					
	1 Dockers	2 Dockers	3 Dockers	4 Dockers	5 Dockers	6 Dockers	1 Dockers	2 Dockers	3 Dockers	4 Dockers	5 Dockers	6 Dockers
1 Dockers		L -0,72	L -0,857	L -0,902	L -0,924	L -0,944		L -0,743	L -0,862	L -0,904	L -0,928	L -0,946
2 Dockers	p 0,000112		M -0,371	L -0,558	L -0,642	L -0,717	p 0,000089		M -0,448	L -0,582	L -0,698	L -0,773
3 Dockers	p 0,000112	p 0,070405		S -0,252	M -0,396	L -0,481	p 0,000114	p 0,048391		S -0,172	M -0,426	L -0,536
4 Dockers	p 0,000082	p 0,013455	p 0,156805		N -0,124	S -0,18	p 0,000082	p 0,009743	p 0,191964		S -0,241	M -0,364
5 Dockers	p 0,000052	p 0,000354	p 0,052504	p 0,614315		N -0,08	p 0,000058	p 0,000172	p 0,042723	p 0,354629		N -0,108
6 Dockers	p 0,000052	p 0,000711	p 0,013455	p 0,152613	p 0,614315		p 0,000058	p 0,000316	p 0,008778	p 0,043083	p 0,354629	

Finally, at 3000 seconds (see Table 4.16), Wilcoxon and Cliff’s delta continue differing from instruction and branch coverage. For instruction coverage, Wilcoxon specify no significant difference for 3 & 4, 4 & 5, and 5 & 6 groups of Dockers. Whereas Cliff’s delta points negligent difference for 5 & 6 groups of Dockers, small difference for 3 & 4, 4 & 5, and 4 & 6, medium difference for 2 & 3 groups of Dockers, and large difference for the other groups.

Then, for branch coverage, Wilcoxon indicates no significant difference only for 3 & 4 and 5 & 6 groups of Dockers. And Cliff's delta points small difference for 3 & 4 and 5 & 6, medium difference for 4 & 5 groups of Dockers, and large difference for the other groups.

Table 4.16 Shopizer significant difference at 3000 seconds
Wilcoxon: p-value (p), Cliff's δ : negligent(N), small(S), medium(M) and large(L) difference

Cliff's δ Wilcoxon	Instruction Coverage						Branch coverage					
	1 Dockers	2 Dockers	3 Dockers	4 Dockers	5 Dockers	6 Dockers	1 Dockers	2 Dockers	3 Dockers	4 Dockers	5 Dockers	6 Dockers
1 Dockers		L -0,645	L -0,851	L -0,893	L -0,93	L -0,915		L -0,674	L -0,882	L -0,913	L -0,94	L -0,928
2 Dockers	p 0,001319		M -0,446	L -0,61	L -0,732	L -0,783	p 0,001154		L -0,592	L -0,71	L -0,832	L -0,851
3 Dockers	p 0,000028	p 0,047193		S -0,306	L -0,522	L -0,612	p 0,000025	p 0,01103		S -0,316	L -0,641	L -0,76
4 Dockers	p 0,000028	p 0,003203	p 0,115218		S -0,232	S -0,32	p 0,000025	p 0,000711	p 0,075307		M -0,335	L -0,512
5 Dockers	p 0,000028	p 0,000028	p 0,006422	p 0,183593		N -0,102	p 0,000025	p 0,000025	p 0,001546	p 0,049846		S -0,221
6 Dockers	p 0,000028	p 0,000172	p 0,007976	p 0,047193	p 0,673247		p 0,000025	p 0,000041	p 0,002303	p 0,004922	p 0,265425	

Related to Shopizer resource consumption, Table 4.17 presents the CPU and memory consumption average of all groups of Dockers. CPU and memory consumption are increased around 4–7% and 6–8% respectively, for each new distributed Docker instance executed.

Table 4.17 Shopizer Consumption Average

Dockers	1	2	3	4	5	6
CPU	19.17%	26.08%	31.53%	36.29%	42.28%	46.14%
MEM	32.06%	38.88%	46.29%	52.9%	60.4%	67.3%

Answer to the research question

Statistical analysis of Parabank and Shopizer results allow us to answer *Distributed-RQ1*: *Does a distributed architecture allow scriptless testing tools to speed up state model inference?* Kruskal Wallis, Wilcoxon and Cliff's delta tests demonstrate that using a distributed approach is significantly different throughout the state model inference process depending on the number of distributed instances used. Therefore, we can reject the *Distributed* – H_0 : *The distributed approach does not reduce the time required to infer a state model.*

4.1.5 Threats to validity

This subsection mentions some threats that could affect the validity of the results [297, 232, 85].

Face validity

We use code-level metrics to evaluate the efficiency of the distributed approach. Although these metrics have wide adoption in the GUI testing field for desktop and mobile applications [220, 266], they seem unusual in web domain GUI testing studies [166], apparently due to the lack of access to the server-side services [66].

Content validity

Compared with other state-of-the-art research, this GUI web study focuses its empirical evaluation using code-level metrics. In future GUI studies, we will research the additional usage of model-level and GUI-level metrics [66].

This study evaluates whether the distributed approach allows scriptless testing tools to speed up state model inference. We consider that measuring code and GUI coverage within a given time budget determines the efficiency of the distributed approach. However, the results of this experiment can be considered as an evaluation of efficacy rather than efficiency, as it measures the extent of coverage achieved in a limited timeframe without analyzing the total required time to explore all potential states.

Internal validity

We detected that some distributed instances stopped receiving messages from the chromedriver renderer, which caused some Docker groups to decrease the number of working instances. These groups of Dockers were re-executed.

We decided to start extracting coverage metrics before building and deploying all involved software in the architecture of the experiment. Something that we consider to be closer to the industrial pipelines. Although this has disrupted the first seconds of coverage measurement, the subsequent coverage growth has shown that a distributed architecture improves according to the number of instances used despite having a different start time.

External validity

Two web applications with a DMSS architecture were used to conduct the study, which accomplishes an essential part of today's web services, sharing resources [62]. Even though we demonstrated the distributed approach helps improve the model inference efficiency, and we faced relevant GUI model inference challenges, to generalize our results properly, we will conduct future experiments with different types of desktop, web, and mobile systems.

Conclusion validity

Because there is a certain degree of randomness when choosing the actions to execute, we cannot assume normal distribution in the experiments [18]. To deal with it, we have applied Kruskal Wallis, Wilcoxon, and Cliff’s delta statistical non-parametric tests to the results obtained from the 30 runs of all Dockers groups.

This study uses an alpha level of 0.05 to determine whether a statistical test indicates a significant difference between the distributed scriptless instances. However, because this evaluation includes two statistical tests—Kruskal Wallis and Wilcoxon—to determine significance, there is a risk of false positives due to multiple comparisons. The absence of a Bonferroni correction [21], which would adjust the alpha level to 0.025 (0.05 divided by 2), should be considered a potential threat to the validity of the conclusions.

Credibility

JaCoCo is used to obtain code coverage metrics. Since JaCoCo is an open-source project widely adopted by the software research and industry development communities, we rely on its definitions of instruction and branch coverage and precise measurements [125].

4.2 Summary

This chapter presents a new technique that improves the speed of model inference in automated scriptless GUI testing, namely distributed state model inference. The DMSS architecture allows multiple instances to infer a shared state model. The proposed shared knowledge algorithm allows these distributed instances to coordinate and improve the inference speed by dividing the unvisited actions to be explored. Moreover, we have explained the abstraction challenges encountered, related to dynamism and non-determinism, and our solutions to reduce the impact on the distributed model inference.

To validate the distributed approach, we prepared an empirical experiment with two different open-source web SUTs. The configuration of independent variables for each SUT is required for the model inference with a single instance or distributed approach. On the one hand, the results of our experiment indicate that a DMSS architecture improves the speed of the state model inference process. On the other hand, it requires investing additional resources to prepare the distributed testing architecture and increase the hardware consumption. With the results presented in this chapter, companies can decide if increasing resources to improve the inference speed is beneficial.

Chapter 5

TESTAR: Online test oracles

A *test oracle* is a mechanism that determines whether a given test activity sequence is an acceptable behavior of the System Under Test (SUT) or not [28]. In scriptless testing, the test activity sequence is generated, one step at a time, through interactions with the SUT by executing actions. In particular, test oracles are defined to evaluate each state visited by the TESTAR tool. Then, TESTAR applies verdicts to these states to ascertain whether the test oracles have passed or failed.

The evaluation of the states discovered by TESTAR can be performed using *online* (or *on-the-fly*) oracles, as well as with *offline* oracles. Online oracles evaluate each state of the SUT during the testing process at runtime. In contrast, offline oracles perform their evaluation asynchronously after testing, using extracted and stored data. In the TESTAR tool, the state model can be leveraged to define and apply these offline oracles [73].

An *implicit test oracle* relies on general, implicit knowledge to distinguish between a system's correct and incorrect behavior [28]. In TESTAR, three default implicit online oracles aim to detect the violation of general-purpose system requirements, such as:

1. Detecting that the SUT *unexpectedly closed* due to a system crash.
2. Detecting that the SUT becomes *unresponsive* due to a freeze.
3. Detecting that any of the GUI widgets contain a *suspicious message* in some of their properties.

Since test oracles are not universal for every SUT—because different systems may consider a behavior acceptable that is considered a failure in another system (e.g., displaying an exception message to the user in the GUI to inform about a software failure can be considered a correct debug behavior in an IDE but a failure in a banking application used by a final customer)—TESTAR allows to enable, disable, and configure these default online test

oracles. Furthermore, TESTAR allows users to define more sophisticated application-specific test oracles by programmatically adding the desired SUT-specific oracles in the TESTAR Java protocols.

In previous TESTAR research, the three default implicit online oracles and new SUT-specific implicit online oracles were configured to successfully detect software failures. The tool was able to discover diverse crash and freeze failures in well-known MacOSX and Windows desktop applications such as Word, PowerPoint, Excel, Skype, and iTunes [33, 35, 13, 80–82]. In collaboration with Clavei [32], SOFTEAM [36], Berner & Mattern [277], Ideanova [172], and Capgemini and ProRail [54], TESTAR was extended with suspicious message test oracles for detecting error and exception messages. Additionally, SUT-specific test oracles were investigated, such as Spanish localization issues, abnormal dialog representations on the screen, and connections with the web server’s logging file to detect missing resources and check the consistency of the SUT database. Furthermore, the tool explored the integration of a REST API to automate the testing of various Internet of Things (IoT) devices in a smart home system, demonstrating the customizability of the test oracles by detecting IoT communication faults [173].

In recent TESTAR research, offline oracles were applied to test for accessibility properties using the WCAG2ICT guidelines [145, 73]. For example, the Navigable Guideline indicates that the SUT should provide ways to help users navigate, find content, and determine their location. An offline oracle can partially test this behavior by checking for ambiguous window titles. If there are too many window widgets with duplicated titles, the offline test oracle must indicate a potential accessibility failure.

This chapter aims to evaluate and enhance the existing test oracles for online exploration. Then, subsequent Chapter 6 presents an offline delta test oracle implementation using inferred state models for detecting GUI delta changes.

In Section 5.1, we describe the online oracles built into TESTAR, which cover the detection of suspicious messages using regular expressions, test oracles designed to identify GUI failures related to SUT functionalities, GUI visual failures, security flaws, and highlight the importance of actions in potentially triggering these failures. In Section 5.2, we present online oracles for XR game systems. In Section 5.3, we present how TESTAR execution test results are created. Finally, in section 5.4, we summarize the outcomes of these online test oracles.

5.1 Online Graphical User Interface test oracles

The TESTAR test oracles that verify whether the SUT process of desktop applications continues to run without crashing or freezing during GUI interactions have been extended for web and mobile systems. These oracles now ensure that webpages or mobile app activities remain active and that the widget tree can be retrieved within a specified time frame. These implicit test oracles help ascertain the robustness of the SUT during scriptless exploration. However, they lack the flexibility for further extension and improvement. Therefore, in this section, we focus on the flexible test oracle mechanism provided by TESTAR, which allows configuring the tool to detect functional, visual, and security software failures.

5.1.1 Suspicious messages regular expressions

The intrinsic objective of detecting suspicious messages is simple yet powerful and flexible. The suspicious messages oracle aims to identify sequences of characters or patterns that can be considered suspicious to be a failure in a specific SUT. For example, when testing a banking GUI application from the user's perspective, it is crucial to avoid displaying any "Error" or "Exception" messages in the GUI that might cause users to worry about the system's behavior.

To configure suspicious messages, the TESTAR tool allows the use of regular expressions (regex) to customize suspicious characters or patterns and select the widget properties (associated widget Tags) to be examined. For instance, Example 5.1 demonstrates how the regular expressions for messages considered suspicious (`.*[eE]rror.*|.*[eE]xception.*`) are configured to be found in the GUI, customized for the widget Tag properties of desktop (Title), web (WebTextContent), and mobile (AndroidText) systems (which are the defined API-Taggable final classes UIATags, WdTags, and AndroidTags). Figure 5.1 shows an example of a GUI error message TESTAR can detect with the GUI suspicious messages test oracle.

```
SuspiciousMessage = .*[eE]rror.*|.*[eE]xcepti[o?]*n.*  
TagsForSuspiciousOracle = Title;WebTextContent;AndroidText
```

Example 5.1 Regular expression for detecting suspicious messages in GUI widgets

The suspicious messages test oracle has proven effective in discovering SUT failures with regular expressions. These regular expressions were easily configured for GUI widget properties. However, as demonstrated in collaboration with SOFTEAM, the suspicious messages oracle can also be applied at other system levels, such as SUT logs [36]. For this



Fig. 5.1 Error message displayed in the GUI of the UPV

reason, TESTAR has been extended to apply suspicious regex messages to the output and error buffers of running SUT processes and the browser's console where web SUTs are running.

Software applications can experience crashes or freezes due to internal code bugs that throw exceptions. Modern software and programming languages include mechanisms for fault tolerance, ensuring that the SUT can maintain a robust response to internal code exceptions without crashing [96]. These exceptions are often recorded in debugging logs, as seen in the studies with SOFTEAM and TESTONA [36, 277]. However, internal failures can also throw exception messages in the output or error buffer of the SUT process.

In TESTAR, we have integrated a *ProcessListener* feature to monitor the output and error buffers of desktop application processes. This allows users to apply suspicious message regex test oracles. Example 5.2 demonstrates how TESTAR can be configured to enable the *ProcessListener* test oracle feature and customize the detection of suspicious messages using patterns (`. [eE] rror . | . [eE] xception .`). Figure 5.2 shows an example of an error message displayed in the buffer of a desktop SUT process when the application crashes.

```
ProcessListenerEnabled = true
SuspiciousProcessOutput = .*[eE] rror .*|. *[eE] xcep[ ct] i[o?]n.*
```

Example 5.2 Regular expression for detecting suspicious messages in output and error buffers of a SUT process

Similarly, suspicious messages often appear in the browser console due to JavaScript exceptions or warnings. Selenium WebDriver can manage these console messages as log activity. Therefore, we have integrated these capabilities in TESTAR to apply suspicious message regex in the browser console when testing web SUTs. Example 5.3 demonstrates

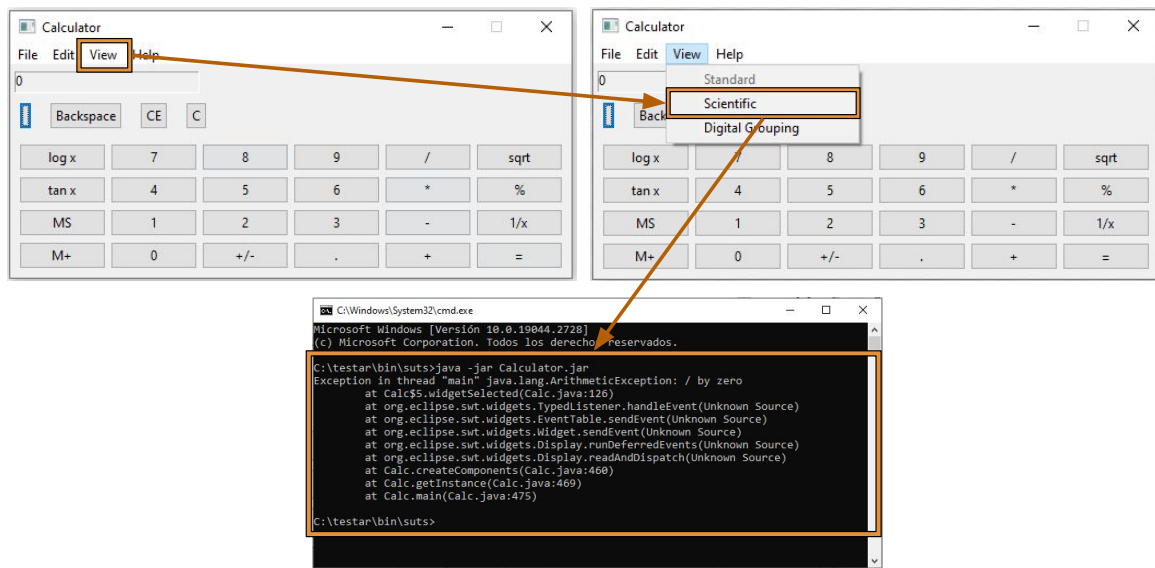


Fig. 5.2 Error message displayed in the buffer of a SUT process

how to configure TESTAR to detect all suspicious message patterns (. * . *) at the error and warning levels in the browser console. Figure 5.3 illustrates two JavaScript error messages that TESTAR can detect using the web browser console's suspicious message test oracle.

```
WebConsoleErrorOracle = false
WebConsoleErrorPattern = .*.*
WebConsoleWarningOracle = false
WebConsoleWarningPattern = .*.*
```

Example 5.3 Regular expression for detecting suspicious messages in the warning and error web console

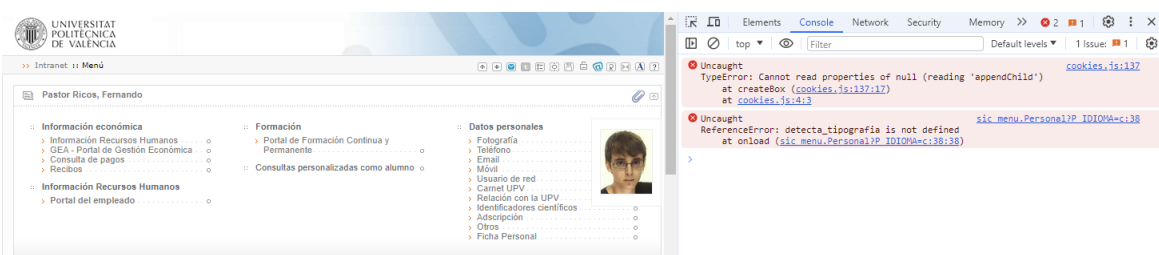


Fig. 5.3 JavaScript error messages displayed in the browser console of the UPV

5.1.2 Graphical User Interface programmatic test oracles

TESTAR enables the programmatic extension of desktop, web, and mobile Java protocols to customize test oracles for specific SUT requirements or scenarios. These programmatic application-specific test oracles can be integrated as verdict conditionals to verify widget properties for missing descriptions [247], detect visual display failures [32], or establish connections with external services or databases [36]. Listing 5.4 demonstrates how to programmatically extend the TESTAR Java protocol to integrate a web verdict that detects select web elements that contain zero or one item.

```

1  @Override
2  protected Verdict getVerdict(State state) {
3      // Invoke integrated implicit oracles to detect
4      // System crashes, non-responsiveness, and suspicious messages
5      Verdict verdict = super.getVerdict(state);
6
7      // Join the default verdict with custom SUT-specific test oracles
8      verdict.join(detectEmptySelectItems(state));
9
10     return verdict;
11 }
12
13 // Detect dropdown select web elements that contain zero or one item
14 public Verdict detectEmptySelectItems(State state) {
15     for(Widget w : state) {
16         if(w.get(Tags.Role).equals(WdRoles.WdSELECT)) {
17             String query = String.format("return_document.getElementById
18             ('%s').length", w.get(WdTags.WebId));
19             Long itemsLength = (Long) WdDriver.executeScript(query);
20             // If this web select element contains zero or one items
21             if (itemsLength.intValue() <= 1) {
22                 // Return a warning verdict
23                 String msg = "Empty_or_Unique_Select_element_detected!";
24                 return new Verdict(Verdict.SEVERITY_WARNING, w, msg);
25             }
26         }
27     }
28     // If all web select elements contain more than one item, return OK
29     return Verdict.OK;
30 }

```

Listing 5.4 Integrating a GUI verdict in the TESTAR Java protocol

Through collaborations with various companies and stakeholders, various software failures that may be common in industrial applications were identified, and test oracles were implemented in TESTAR to detect these issues during scriptless exploration.

Data display fault

GUI widgets are essential for displaying software data to users and providing them access to complex systems. However, this data can be vulnerable to storage or software logic bugs, which may cause the SUT to exhibit various data display faults. Invariant test oracles [180], designed to verify the invariant properties of widgets, can be integrated into TESTAR to identify these SUT faults effectively. For instance, the protocol outlined in Listing 5.4 demonstrates how to check for empty values in dropdown lists. Additionally, this protocol can be extended to detect other invariant property violations, such as improperly sorted or duplicated elements in dropdown lists and tables, formatting errors like monetary values with more than two decimal places or dates with less than six digits [150], and the presence of unintended HTML or XML tags, which can lead to user misinterpretation.

Figure 5.4 shows an Amazon product page where users can select various laptop RAM and disk GB capacity options. On the left side of the figure, the product description values match the selected options. Nevertheless, in the right-side example, the storage GB value does not match the selected option. Figure 5.5 illustrates another issue with a different Amazon product. The table describing the specs of the graphics card contains a language row where the value "English" is repeated multiple times. Finally, Figure 5.6 exemplifies a reported and fixed duplicated element issue in the OBS desktop application¹.

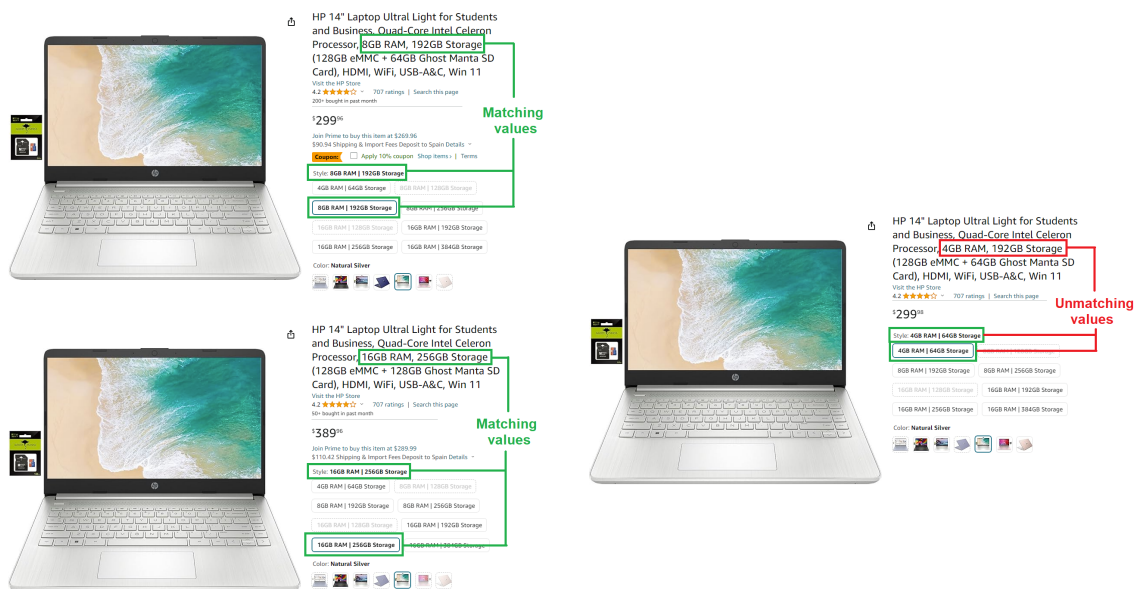
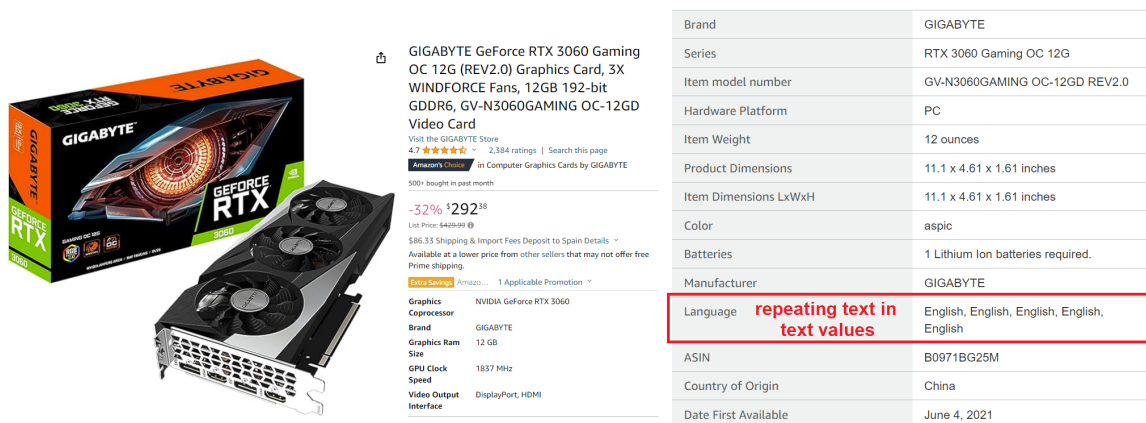


Fig. 5.4 Unmatching values GUI failure in an Amazon web page

¹<https://github.com/obsproject/obs-studio/issues/6914>



Brand	GIGABYTE
Series	RTX 3060 Gaming OC 12G
Item model number	GV-N3060GAMING OC-12GD REV2.0
Hardware Platform	PC
Item Weight	12 ounces
Product Dimensions	11.1 x 4.61 x 1.61 inches
Item Dimensions LxWxH	11.1 x 4.61 x 1.61 inches
Color	aspic
Batteries	1 Lithium Ion batteries required.
Manufacturer	GIGABYTE
Language	repeating text in text values English, English, English, English, English
ASIN	B0971BG25M
Country of Origin	China
Date First Available	June 4, 2021

Fig. 5.5 Repeated text GUI failure in an Amazon web page

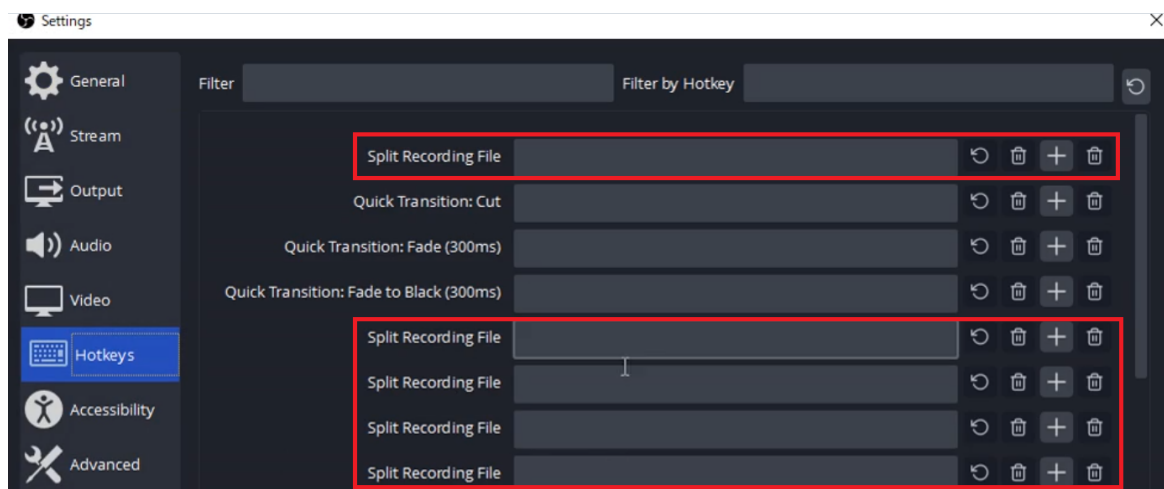


Fig. 5.6 OBS Issue: "Split Recording File" hotkey increases when changing profiles

While these real GUI failures might be considered low severity in these specific examples, they could have serious implications in critical systems, such as hospitals. In such a scenario, a similar GUI failure might indicate that an internal value related to a patient's condition or medication is not correctly assigned within the system. Furthermore, applying this test oracle to thousands of products in applications like Amazon would be impractical with manual or scripted approaches due to the sheer number of product paths that would need to be tested. This highlights the importance of an automated and scalable testing approach like the TESTAR scriptless testing exploration approach to evaluate these possible types of software failures.

Misspelling issues

Software applications designed to provide information in various languages or dialects often encounter significant spelling challenges in translation accuracy and grammatical correctness [46]. For instance, Figure 5.7 shows two misspelling issues related to an accent character and English translations on the UPV web page. Previous studies configured TESTAR to identify localization issues through suspicious messages regex expressions [172]. Nevertheless, for large SUTs, this configuration demands extensive prior knowledge and significant effort to account for all possible misspellings.

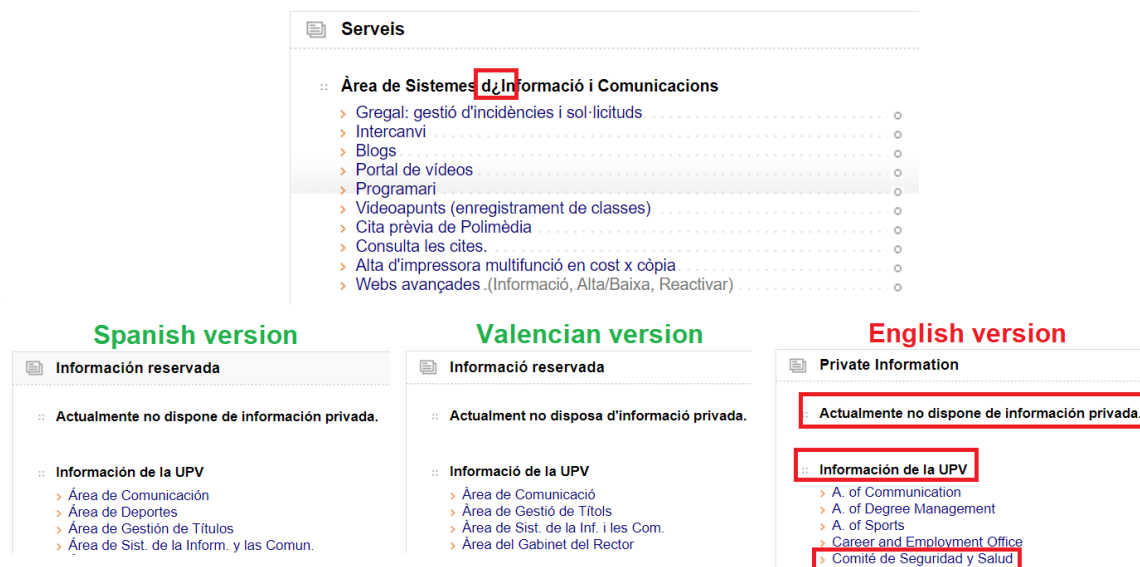


Fig. 5.7 UPV misspelling issues related to an accent character and English translations

The JLanguageTool² library has been integrated into TESTAR, a free grammar checker that supports more than 30 languages [187]. This allows users to enable scriptless exploration with a built-in spell-checking test oracle, though some effort is still needed to specify words to be ignored. Example 5.5 showcases a Java method that utilizes JLanguageTool to check for spelling rule violations in a specified language. The user can specify the widget Tag property to which to apply the spell checking, the language for the check, and a file or regex pattern that defines the spelling rules to ignore.

Visual failures

Visual or layout GUI failures are another common issue in desktop, web, or mobile applications. These failures can be related to blurry images or image resources that fail to load or

²<https://languagetool.org/>

```

1  public static Verdict SpellChecker(State state, Tag<String> tagChecker,
    Language languageChecker, String ignoreListFile, String ignoreRegex)
    {
2  loadSpellingIgnoreListFromFile(ignoreFile); // Load ignore list
3  Verdict spellCheckerVerdict = Verdict.OK;
4  Pattern ignorePattern = Pattern.compile(ignoreRegex);
5  JLanguageTool langTool = new JLanguageTool(languageChecker);
6
7  for (Widget w : state) {
8      String tagTextContent = w.get(tagChecker);
9      // Check the loaded file to determine if the text should be ignored
10     if (isNotOnIgnoreList(tagTextContent)) {
11         // Check the pattern to determine if the text should be ignored
12         Matcher ignoreMatcher = ignorePattern.matcher(tagTextContent);
13         if (ignoreRegex.isEmpty() || !ignoreMatcher.find()) {
14             List<RuleMatch> matches = langTool.check(tagTextContent);
15             for (RuleMatch match : matches) {
16                 String misspellingMsg = "Potential_mispelling_" + match.
                    getFromPos() + "-" + match.getToPos() + ":_:" + match.
                    getMessage();
17                 String correctMsg = "Suggested_correction(s):_" + match.
                    getSuggestedReplacements();
18                 String verdictMsg = "Widget_Title_(_" + tagTextContent + "_)_"
                    + misspellingMsg + "_" + correctMsg;
19                 // Join the misspelling Verdict and continue with next widgets
20                 spellCheckerVerdict = spellCheckerVerdict.join(new Verdict(
                    WARNING_SPELLING_ISSUE, verdictMsg));
21                 // Save the misspelling to the ignore list to avoid duplicates
22                 saveMisspellingIntoIgnoreListFile(ignoreListFile,
                    tagTextContent);
23             }
24         }
25     }
26 }
27 return spellCheckerVerdict; // Return none, one, or multiple
    misspelling issues
28 }

```

Listing 5.5 A TESTAR example of a spell-checker test oracle that uses JLanguageTool

are missing [161], misaligned widgets such as text and buttons that conform to a form due to inconsistent margins and padding [37], overlap and clash of widgets due to the design of a poor layout that doesn't adapt to different screen sizes and orientations [108, 233], and more.

Various techniques are being integrated into TESTAR to detect visual and layout GUI failures. One method involves using the coordinates of leaf widgets in the state to identify clashes between widgets from different sub-trees. Another approach incorporates balance, symmetry, and alignment metrics to evaluate GUI aesthetics [305]. Figure 5.8 is not a failure

detected by TESTAR, but shows a real visual failure reported and fixed clash detection issue between two widgets in the OBS desktop application³.

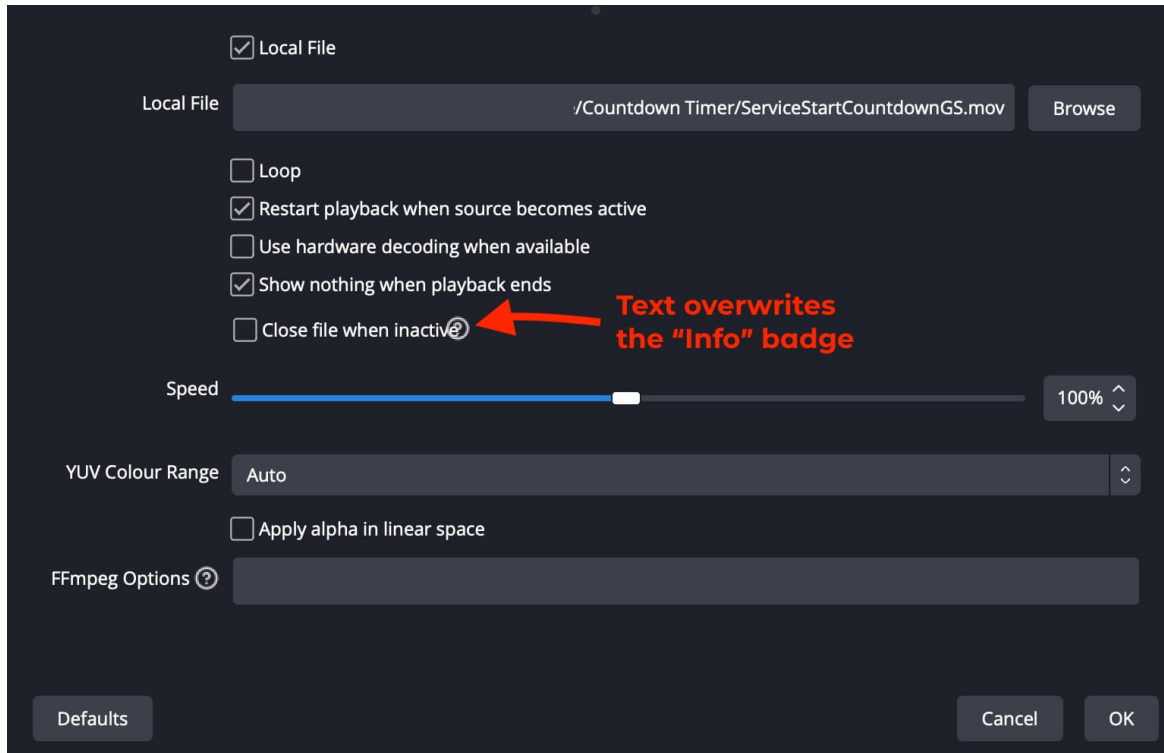


Fig. 5.8 OBS Issue: Dialog box text overwrites the Info badge

5.1.3 Security web test oracles

Software bugs, whether introduced intentionally or unintentionally, can cause the software to fail if its features deviate from its intended behavior or do anything unintended [257]. In terms of security, these faults are referred to as security flaws—defects in the software that create vulnerabilities exploitable by malicious attacks. As the world rapidly adapts to the digital environment by adopting technologies such as cloud computing, the Internet of Things (IoT), social media, and wireless communication, the potential for introducing vulnerabilities—and thus opportunities for malicious attacks—increases significantly [135, 22].

Software security focuses on ensuring that software behaves correctly in the presence of malicious attacks [222]. The primary objective of security testing is to verify and validate software systems concerning security properties such as confidentiality, integrity, availability, authentication, authorization, and nonrepudiation [84]. Static analysis techniques, such as

³<https://github.com/obsproject/obs-studio/issues/7758>

vulnerability code scanning and security code reviews, analyze the source code or compiled code to detect potential security flaws early in the development process [84]. Dynamic analysis techniques, like network vulnerability scanners and fuzz testing, interact with the SUT at runtime without requiring access to the source code.

During the last few years, dynamic analysis techniques have been introduced into the TESTAR tool to enable the definition of test oracles capable of detecting potential web SUT vulnerabilities [118, 43]. Selenium WebDriver incorporates the DevTools API, which provides access to the browser's network interface. This interface enables TESTAR to enrich the GUI state information with network data, including HTTP headers, cookie configurations, and the usage of integrated third-party libraries, APIs, or scripts.

HTTP Headers Scanner

The Network API⁴ enables a DevTools extension to gather information about network requests associated with the web SUT that the DevTools are monitoring. TESTAR can utilize this API to retrieve HTTP header information whenever the web SUT responds to a triggered event, such as a performed GUI interaction. Due to the architecture or implementation of the web SUT, it is possible to receive multiple web responses following a single interaction or because of the dynamic nature of the web SUT's network connections.

Between the intervals of executed actions, TESTAR stores the retrieved HTTP header information. It then applies a set of test oracles to these headers to verify compliance with specific security rules:

- **Strict-Transport-Security:** Ensure that this header is present to enforce communication using HTTPS instead of HTTP.
- **X-Content-Type-Options:** Verify that this header contains the 'nosniff' flag, which instructs the browser not to attempt to determine the MIME type on its own [29]. Combined with the appropriate 'Content-Type' header, this reduces the risk of users being affected by maliciously injected files, such as web scripts.
- **X-Frame-Options:** Confirm that this header is present to deny or restrict loading frames from different origins [245].
- **X-XSS-Protection:** Ensure that this header includes the '1; mode=block' flag, which will instruct the browser to prevent the page from rendering if an XSS attack is detected [30].

⁴<https://developer.chrome.com/docs/extensions/reference/api/devtools/network>

- **Set-Cookie:** Verify that this header contains the 'secure' flag, indicating that a cookie should only be sent to the web server over HTTPS connections [148].

Cross-Site Scripting

Cross-Site Scripting (XSS) is an attack that aims to inject scripts into web applications, causing victims' web browsers to execute malicious actions such as accessing sensitive resources like cookies, passwords, and personal data [105]. XSS has been recognized as one of the most dangerous threats to web application security since 1999 and has consistently appeared on OWASP's list of top 10 vulnerabilities since 2003 [112]. The need for rapid changes in web services to meet customer needs and the integration of scripting languages in dynamic web applications make web systems continuously potentially vulnerable to XSS attacks.

TESTAR is being enhanced to work as a dynamic exploitation agent designed to detect potential XSS vulnerabilities. To achieve this, TESTAR integrates a customizable input manager that allows to adapt the tool's typed text and characters to various testing scenarios. For XSS detection, we aim for TESTAR to generate type actions that inject XSS payloads during web runtime execution. Subsequently, TESTAR must be able to identify the successful injection of XSS during its exploration process.

To manage a controlled XSS injection that can be detected during execution, TESTAR, by default, uses a few simple variations of scripts designed to print a message in the browser's console. Combined with the existing test oracle feature that monitors for suspicious messages in the console, this approach enables the detection and reporting of XSS vulnerabilities. Example 5.6 illustrates the input text for GUI widgets in the first line and its encoded version for use with URL parameters in the second line. This approach ensures that TESTAR does not interfere with the exploration of the SUT by using more disruptive XSS injections, such as alert pop-ups [133]. Moreover, displaying a message in the console is a non-intrusive XSS attack that does not harmfully affect the SUT.

```
<script>console.log('XSS_detected!');</script>  
<script>console.log(%27XSS%20detected!%27);</script>
```

Example 5.6 Default XSS injection characters used by TESTAR

The TESTAR input manager can be customized, and its Java protocol can be programmatically extended to enable more sophisticated XSS injections. One of our ongoing implementations involves enhancing the exploitation logic to intelligently detect if the web SUT employs any sanitization mechanisms. TESTAR will then attempt various payloads or encode

characters to bypass these sanitization methods. For instance, Figure 5.9 demonstrates that attempting to inject an XSS JavaScript console message using default special characters is denied due to the web SUT character sanitization. However, encoding these characters as HTML codes permits to bypass the sanitization and successfully execute the XSS injection.

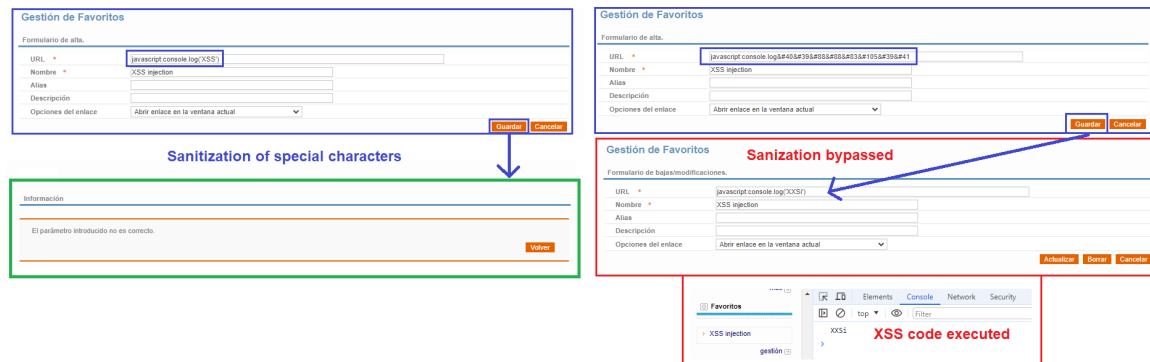


Fig. 5.9 UPV feature that sanitizes special characters but is vulnerable to HTML encoding

SQL injection

SQL injection is an attack that involves injecting a malicious database query into the frontend of a SUT to gain unauthorized access, retrieve, modify, or delete data stored in the backend database. Web systems have been particularly vulnerable to SQL injection attacks, and OWASP has classified it as the third most serious web application security risk in their Top 10 list published in 2021 [68].

Similar to XSS vulnerabilities, we are concurrently enhancing TESTAR to work as a dynamic exploitation agent capable of identifying potential SQL injection vulnerabilities. To manage a controlled SQL injection that can be detected during execution, TESTAR, by default, uses the simple quote ' character for input text fields and the encoded version %27 for URL parameters. The default objective is to provoke the backend SQL server to generate an exception that disrupts the normal flow of the web server's response. This exception might result in an HTTP 5XX server error message, detectable through the Network API of the DevTools extension or by the test oracle feature that monitors for suspicious messages in the browser console.

5.1.4 Evaluation of online GUI oracles

The suspicious messages test oracles continue to demonstrate their effectiveness in discovering SUT failures. In an industrial collaboration with Kuveyt Türk Bank, TESTAR identified two significant errors that would have been visible to end users [5]. Moreover, in a

consortium with Prodevelop, TESTAR successfully detected two known errors related to the SUT database connection [237].

Additionally, in collaboration with a senior tester at DigiOffice company, the potential of TESTAR oracles was analyzed and configured to investigate the tool bug-finding capabilities for real industrial historical bugs [43]. This effort included configuring suspicious message expressions for GUI and browser console script errors; implementing new programmatic GUI test oracles for data display faults, misspelling issues, and visual failures; as well as security web test oracles. This collaboration led to the discovery of more than 50 new verified bugs in DigiOffice.

These industry experiences are described in detail in Chapter 7.

5.2 Online Game test oracles

In game systems, the TESTAR agent can employ implicit test oracles to verify that the game-SUT process continues to run without crashing or freezing during interactions executed through the iv4XR framework. Additionally, suspicious message test oracles can be configured to detect suspicious characters or patterns within the entities that constitute the game state and to analyze exception messages in external logs. However, while these test oracles offer a solid foundation for assessing the robustness of a game, in the game environment, it is of paramount importance to also test the functional aspects of game entities.

5.2.1 Programmatic test oracles for functional game aspects

The game Space Engineers, developed by Keen Software House and GoodAI, includes a suite of 1,322 mandatory regression tests that must be executed before every game release iteration. These regression tests cover various game features, including the astronaut's movements, attributes, item usage, interactions with blocks, as well as graphical and sound aspects. Additionally, the Space Engineers team has documented another 7,484 tests that address the specific attributes of all Space Engineers' blocks, such as their placement, mass, durability, integrity, usage of components for building, etc. Moreover, over the years of game development, the Space Engineers team has maintained a database list of reproducible bugs discovered by both testers and the player community.

Through iterative collaboration, the analysis of the variety of defined tests and documented bugs has been conducted to identify those suitable for integration as test oracles for scriptless exploration. Figure 5.10 exemplifies the test oracles integrated into TESTAR, which include checking that the integrity of all blocks decreases after grinding or shooting or increases after

welding; ensuring that the agent's health, oxygen, hydrogen, and energy are restored when interacting with medical rooms or cockpits; verifying that the jet-pack and the dampeners are not enabled automatically without player activation after entering a cockpit, medical room, or interacting with a ladder; and validating the ability to construct new blocks if the player has the required materials, such as steel plate components in the astronaut's inventory. These game test oracles typically assess the pre- and post-conditions of game entity properties between two observed game states.



Fig. 5.10 Test oracles that check Space Engineers game features

Similar to GUI test oracles, these game test oracles can be programmatically added to TESTAR Java protocols. Listing 5.7 demonstrates how to integrate a game verdict into a TESTAR game protocol. This test oracle aims to check that the integrity of a block decreases after executing a grinder action or that the interacted block does not exist anymore because it was destroyed.


```

1  @Override
2  protected Verdict getVerdict(State state) {
3      // Invoke integrated implicit oracles to detect
4      // Game crashes, non-responsiveness, and suspicious messages
5      Verdict verdict = super.getVerdict(state);
6
7      // Join the default verdict with custom game-specific test oracles
8      verdict.join(checkBlockIntegrity(state));
9
10     return verdict;
11 }
12
13 // Detect that block integrity decreases after a grinder action
14 public Verdict checkBlockIntegrity(State state) {
15     // If TESTAR agent last executed tactical action is of type grinder
16     if(lastAction instanceof NavigateGrinderBlock) {
17         // Check the block widget attached to the grinder action
18         Widget block = lastAction.get(Tags.OriginWidget);
19         Float preIntegrity = block.get(IV4XRtags.Integrity);
20
21         // Try to find the same block ID in the current state
22         for(Widget w : state) {
23             if(w.get(IV4XRtags.ID).equals(block.get(IV4XRtags.ID))) {
24                 Float postIntegrity = w.get(IV4XRtags.Integrity);
25
26                 // If the block integrity didn't decrease
27                 if(postIntegrity >= preIntegrity) {
28                     // Report a block integrity verdict error
29                     String msg = "Block_integrity_didn't_decrease";
30                     return new Verdict(Verdict.INTEGRITY_ERROR, w, msg);
31                 }
32             }
33         }
34         // If the previous block does not exist in the current state,
35         // the block has been destroyed after the grinder action
36         // We will consider this OK by default
37     }
38     return Verdict.OK;
39 }

```

Listing 5.7 Integrating a game verdict in the TESTAR Java protocol

5.2.2 Evaluation of online game test oracles

TESTAR implicit test oracles intended to verify the robustness of the SUT were useful in the initial versions of the IV4XR framework to detect an invalid action request that hangs the framework⁵. Conversely, programmatic test oracles focused on functional game aspects were employed at various levels in Space Engineers to evaluate the effectiveness of the

⁵<https://github.com/iv4xr-project/iv4xrDemo/issues/9>

TESTAR agent in reproducing game bugs [226]. A video of one of these tests can be seen here⁶, where TESTAR explored a small level for approximately three minutes. During this exploration, it identified a jet-pack bug where the jet-pack was unintentionally enabled after interacting with a ladder attached to a spatial vehicle with artificial gravity.

5.3 Test results

TESTAR scriptless exploration runs consist of a series of test sequences, each comprising a defined number of actions. For each executed sequence, TESTAR saves the following information in a directory named with a timestamp and the SUT name:

- **Logs:** These include details of all executed actions, the target widget, and the various state identifiers of the test sequence, along with timestamps that help synchronize results with other applications.
- **Screenshot Images:** Captures of the GUI state after each action in a sequence. These screenshots are generated using the coordinates of the states and widgets obtained through the technical APIs.
- **HTML reports:** Provide detailed information about the derived and executed actions, combining textual information from the technical APIs with visual screenshots to illustrate the sequence steps. The final verdict in the HTML report indicates whether the sequence contains failures or if the execution is considered correct.

Additionally, an index log is created during the first TESTAR run and updated with each sequence execution. This index supports the integration and synchronization of TESTAR with other applications. Timestamps in the index can be used to locate all TESTAR sequences by following the appropriate timestamped directory (see Figure 5.11).

If the state model inference approach of TESTAR is used during online exploration, an analysis mode allows users to load an interactive web graph. Figure 5.12 illustrates how a user can select an inferred state model to view different state model layers. By clicking on the state and action elements, users can access the properties of states, actions, and widgets.

5.4 Summary

This chapter discusses various online test oracles integrated into the TESTAR tool. Regular expressions are employed to detect suspicious messages in GUI properties or SUT logs.

⁶<https://www.youtube.com/watch?v=ho1EMVtr8C4>

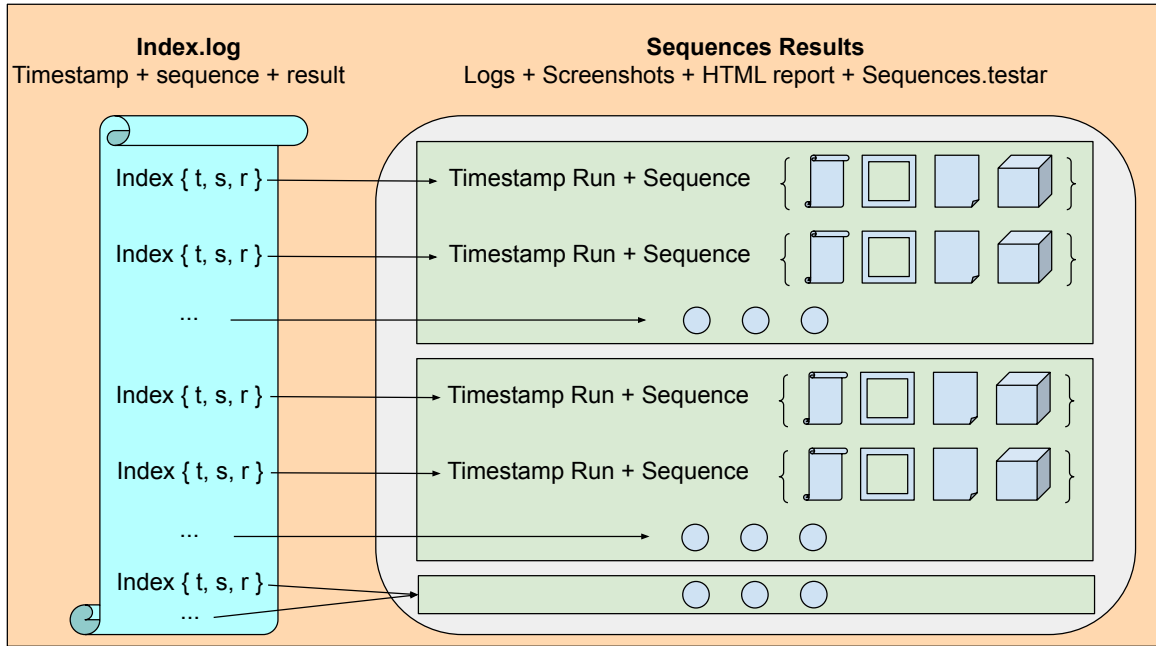


Fig. 5.11 Output Structure for Test Results

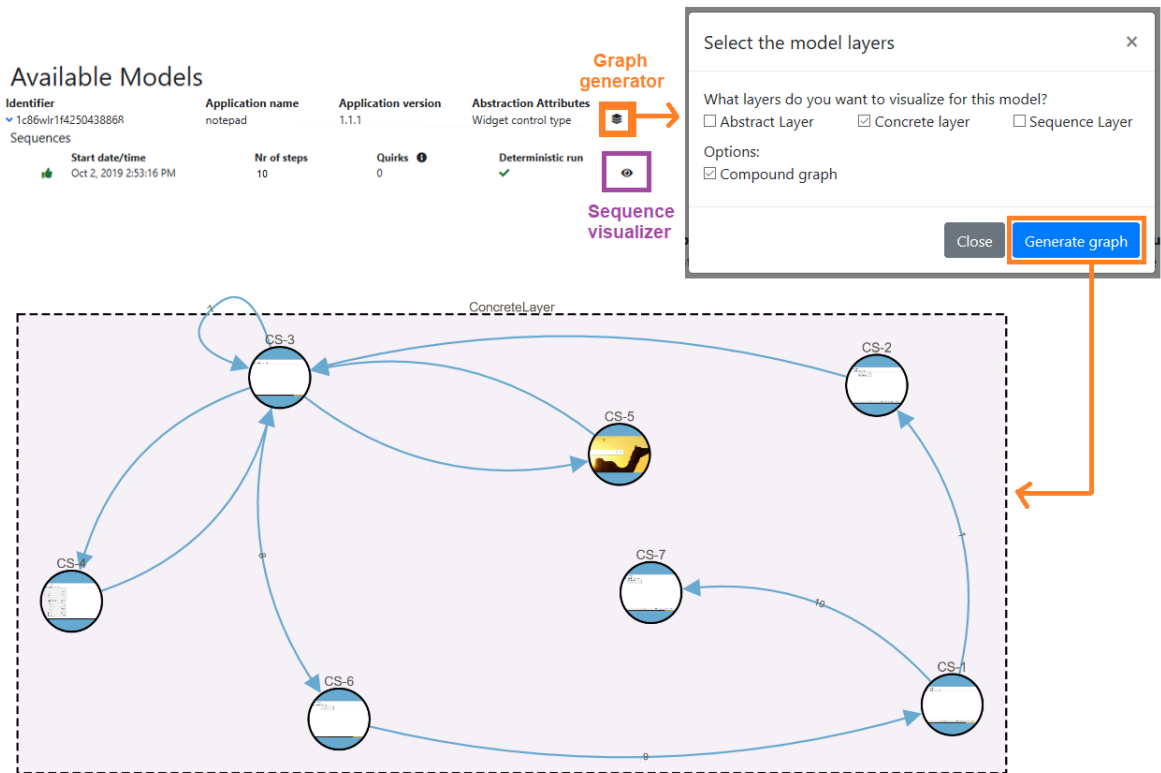


Fig. 5.12 State Model analysis interactive graph

This flexible mechanism has been extended to detect web console exceptions and warning messages and remains as the primary method for identifying SUT failures in GUI systems.

A diverse set of programmatic test oracles has been researched and integrated into TESTAR as GUI test oracles. These oracles address GUI data display faults, misspelling issues, and visual failures. In contrast, for game systems, programmatic test oracles primarily focus on assessing specific functional aspects of the game. This includes checking pre- and post-conditions of game entity properties after performing a game interaction.

Lastly, security test oracles are being incorporated into TESTAR to identify potential security vulnerabilities. Integration with technical APIs, such as the DevTool Network API, allows TESTAR to analyze HTTP header information for web systems. Additionally, combining XSS and SQL injection characters with regular expressions during Type actions helps in detecting XSS and SQL injection vulnerabilities.

Chapter 6

TESTAR: Offline delta test oracles

In recent decades, iterative and incremental software development methodologies have gained widespread acceptance due to their ability to accelerate the dynamic and rapid evolution of software [209, 117, 97]. Across a wide range of projects, from those in small and large industries to personal and research initiatives, there is a high emphasis on facilitating the seamless integration of developers' changes into shared software repositories. These repositories serve as the foundation for building every commit, running tests, documenting requirements, and evaluating the overall quality of the software code [275].

The concept of *delta change* in software development refers to any modification or update from one software version to the subsequent one, regardless of the scale of these changes [249]. These incremental changes, known as delta increments, can involve tasks such as developing new functionalities, fixing bugs, integrating unit tests, or updating documentation. However, these delta increments, whether they introduce, remove, or update software functionalities, may unintentionally introduce failures. Therefore, it is of paramount importance to implement testing processes that focus on detecting, controlling, and minimizing these potential delta failures.

Depending on the size of the delta increment, the testing process may require different types of analyses and verification techniques (see Figure 6.1). Minor delta increments, which involve small changes in code methods, may be adequately validated through software compilation and unit testing. On the other hand, larger increments, such as new delta versions of a software product, demand additional analysis and verification by multiple collaborators and techniques on the project, including usability, functional, and security testing.

As code undergoes continuous evolution via delta increments, techniques such as code reviews have become a standard practice for ensuring readability and consistency in software projects, but also standing as one of the most effective means to uncover bugs [304, 250, 251]. However, the analysis and validation of delta changes within the Graphical User Interface

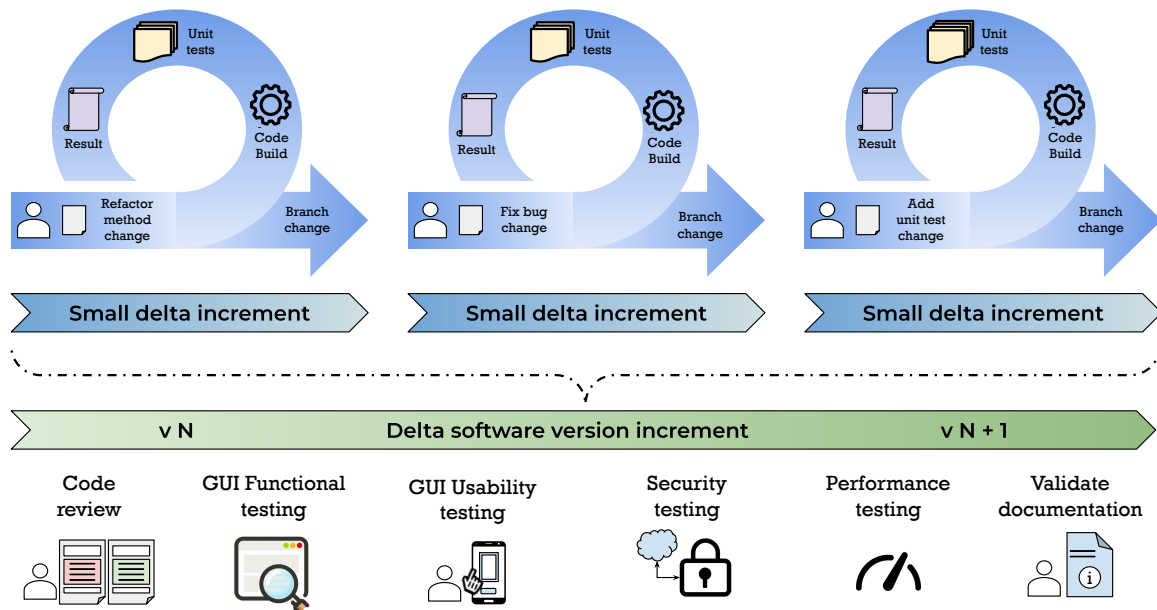


Fig. 6.1 Overview of a delta testing process in software projects

(GUI) remains an underutilized process across desktop, web, and mobile applications. This oversight is unfortunate, given that the GUI serves as the primary point of contact between users and the software system. Overlooking GUI delta changes neglects a fundamental aspect of software quality, potentially impacting the user experience and the overall effectiveness of the application.

We consider that integrating delta change detection techniques to visualize, analyze, and review GUI delta changes stands as a significant step towards comprehensive software quality assurance. By automating this approach, developers, testers, and other project contributors can streamline their efforts in identifying functionalities that have been removed, added, or modified. The information provided by the automated technique serves as a validation step, ensuring that the implemented code adequately reflects the intended GUI changes or uncovers unexpected GUI modifications that can be considered bugs.

In this chapter, we have studied the existing *GUI change detection* practices and developed a novel tool to detect and highlight delta GUI changes of one desktop, web, and Android open-source applications [214]. Section 6.1 reports a systematic mapping of the literature that studies the fundamental concepts, methods, and technologies associated with GUI change detection. Section 6.2 presents a GUI change detection tool that compares and highlights the GUI changes between different versions of desktop, web, and Android GUI systems using TESTAR inferred state models. Finally, Section 6.3 summarizes the GUI change detection study results.

6.1 Systematic mapping of the literature

The continuous evolution of concepts and terminologies related to *Delta GUI Change Detection* in the GUI testing field necessitates a comprehensive understanding of the existing research literature. To address this need, a systematic mapping of the literature was conducted, following the methodological guidelines of Kitchenham et al. [141, 143]. This study was guided by the following research question:

- Delta-RQ1: What techniques are employed for delta GUI change detection, and for which type of systems?

6.1.1 Methodology

Scopus was chosen as the primary database. In comparison with other scientific repositories, such as Web of Science (WoS), Scopus offers broader and more inclusive content coverage [241], ensuring access to an extensive range of relevant research. Furthermore, Scopus provides robust impact indicators that are less susceptible to manipulation and are available for all serial sources in all disciplines [224].

```
TITLE-ABS-KEY(((gui OR ui OR "graphical user interface" OR "user interface") W/2 chang*) AND detect*)
OR TITLE-ABS-KEY((detect* W/2 (gui OR ui OR "graphical user interface" OR "user interface")) AND chang*)
OR TITLE-ABS-KEY((chang* W/2 detect*) AND (gui OR ui OR "graphical user interface" OR "user interface"))

OR TITLE-ABS-KEY(((gui OR ui OR "graphical user interface" OR "user interface") W/2 chang*) AND delta)
OR TITLE-ABS-KEY((delta W/2 (gui OR ui OR "graphical user interface" OR "user interface")) AND chang*)
OR TITLE-ABS-KEY((chang* W/2 delta ) AND (gui OR ui OR "graphical user interface" OR "user interface"))

OR TITLE-ABS-KEY(((gui OR ui OR "graphical user interface" OR "user interface") W/2 chang*) AND (evolv* OR evolut*))
OR TITLE-ABS-KEY((evolv* OR evolut*) W/2 (gui OR ui OR "graphical user interface" OR "user interface")) AND chang*)
OR TITLE-ABS-KEY((chang* W/2 (evolv* OR evolut*)) AND (gui OR ui OR "graphical user interface" OR "user interface"))

OR TITLE-ABS-KEY(((gui OR ui OR "graphical user interface" OR "user interface") W/2 chang*) AND report*)
OR TITLE-ABS-KEY((report* W/2 (gui OR ui OR "graphical user interface" OR "user interface")) AND chang*)
OR TITLE-ABS-KEY((chang* W/2 report*) AND (gui OR ui OR "graphical user interface" OR "user interface"))

OR TITLE-ABS-KEY(((gui OR ui OR "graphical user interface" OR "user interface") W/2 differen*) AND detect*)
OR TITLE-ABS-KEY((detect* W/2 (gui OR ui OR "graphical user interface" OR "user interface")) AND differen*)
OR TITLE-ABS-KEY((differen* W/2 detect*) AND (gui OR ui OR "graphical user interface" OR "user interface"))

AND ( LIMIT-TO ( LANGUAGE , "English" ) OR LIMIT-TO ( LANGUAGE , "t PUBYEAR < 2023" ) )
AND ( LIMIT-TO ( SUBJAREA , "COMP" ) OR LIMIT-TO ( SUBJAREA , "ENGI" ) )
```

Fig. 6.2 Search query to embrace diverse terminology for GUI delta changes

Initially, a search query was formulated to retrieve studies the studies related to the delta GUI change detection approach, which encompass the initial terms delta, GUI, change, and detection. However, insights gained from previous research [239] highlighted the limitations of these terms, as other studies often employ different terminology to disseminate the use of GUI change detection techniques. Consequently, the query was extended by incorporating the term *evolve* to capture descriptions of releasing changes as new application versions, *report* to signify the process of informing and presenting change results, and

difference to address methods capable of identifying and detecting changes. Additionally, recognizing that these terms can be combined with versatile interrelationships, multiple OR disjunction combinations were employed to ensure a comprehensive search. The final search query formula is shown in Figure 6.2:

- The Scopus operator **TITLE-ABS-KEY** specifies that the terms in the query should appear in the title, abstract, or keywords of the scientific publication.
- The term `gui` is expanded to include `ui`, `graphical user interface`, and `user interface` in order to encompass all references to the system's user interface.
- The term `evolve` is expanded to include `evolution`.
- The wildcard character `*` is used for the terms `detect*`, `chang*`, `evolv*`, `evolut*`, `report*`, and `differen*`, allowing for variations in word endings and enhancing the search's inclusivity.
- This query generates multiple combinations of terms to explore versatile interrelationships between them, reducing the likelihood of missing relevant papers.
- The `W/2` operator sets the minimum distance between related terms to 2 words, ensuring a reasonably close relationship between them in the document.
- The query restricts results to publications in the English language.
- The scope focuses on publications in the subject areas of Computer Science (COMP) and Engineering (ENGI).

The search was not limited to specific years to ensure the retrieval of all existing studies. The Scopus search retrieved a total of 820 papers in the range from 1985 to September 2023. Then, to identify relevant research papers within the field of GUI software testing, more specifically about GUI change detection approaches, an iterative exclusion process was established, as depicted in Figure 6.3 and explained below.

1. **Exclusion Based on Field Relevance (Abstract):** In the first step of the exclusion criteria, abstracts were reviewed to remove papers not directly related to the field of GUI software testing. This meant excluding papers that fell into other domains, such as topology, physics, structural engineering, or medical research. Furthermore, publications that were not conference, workshop, journal publications, or book chapters were excluded. After this initial exclusion, 132 papers remained out of the original 820.

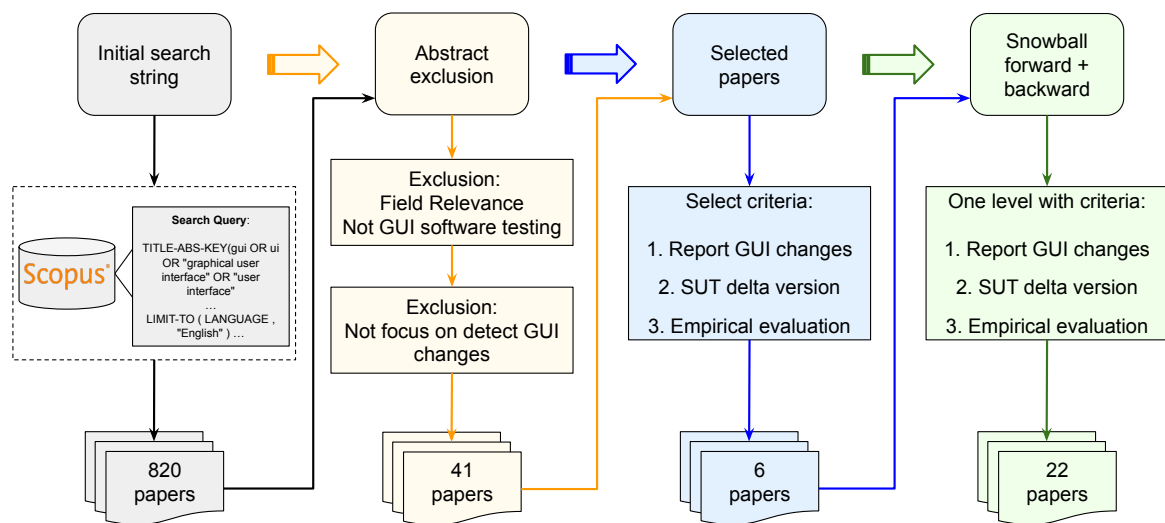


Fig. 6.3 Overview of the process for the systematic mapping literature review

2. **Exclusion Based on GUI Change Detection (Abstract):** The second step involved refining the selection, focusing on papers abstracts related to the detection of GUI changes. This refinement process entailed excluding papers that primarily discussed topics such as identifying GUI state elements or analyzing the aesthetic aspects of GUIs without comparing version changes. As a result, 41 papers were retained from the subset of 132, ensuring a more targeted set of research papers aligned with the study's objectives.
3. **Paper selection based on Report GUI delta changes:** In the third step, a comprehensive reading was conducted with the 41 papers. The focus was on identifying papers that specifically addressed the creation of textual or visual reports to inform about the GUI delta changes between different versions of the System Under Test (SUT). Furthermore, those studies must contain an empirical evaluation that experiments and validates the proposed technique. Papers that exemplify the technique description but do not indicate any evaluation were discarded. As a result, 6 papers, without considering our previous work, were finally selected from the 41 GUI change detection papers of the original group of 820 queried papers.

As an example of an excluded paper, the study by Bures [48] proposes an approach designed to track delta UI changes in software development projects, providing valuable information for test designers. For instance, this information can be employed in the maintenance of test scripts. However, the way in which this information is reported to stakeholders remains unspecified. Furthermore, the simulations or experiments

mentioned appear more as a prospective estimation of the proposal rather than an empirical evaluation.

4. **Snowball paper selection on Report GUI delta changes:** The final step consists of applying one level of backward and forward snowballing process [296]. Since it was determined that relevant GUI change detection techniques could be associated with the subset of 41 papers, the snowball process was applied to the entire subset rather than limiting it to the initial 6 selected papers. The snowball inclusion criteria remain the same as the third step: papers that focus on the creation of textual or visual reports to inform about the GUI delta changes between different versions of the SUT, and that must contain some empirical evaluation that experiments and validates the proposed technique. This snowballing process included 16 additional research papers, which made up a total of 22 GUI change detection papers.

During the snowball process, research on GUI change detection was identified that does not specifically mention or exemplify how changes are reported to users. As an example, the study by Grechanik et al. [100] presents a tool called GUIDE that allows users to differentiate the GUIs of evolving GUI-based application versions. Nonetheless, because the study seems focused on presenting the algorithm improvement, there is a lack of information that indicates how the differencing results are reported to the users.

6.1.2 Data collection

For each selected paper, the whole paper was read and the following information was extracted in an Excel file:

- **Type of SUT:** Identification of the SUT type for which the technique is implemented (desktop, web, mobile, image-based).
- **Change Detection Technique/Algorithm:** Summary of the technique or algorithm employed for GUI change detection (Web Document Object Model (DOM) tree or XPath comparison, Hash calculations and comparison, Visual recognition and comparison techniques, etc.).
- **GUI Data Extraction:** Description of how the GUI information is extracted or utilized to apply the change detection technique or algorithm (Web URL, HTML document, XML data, State screenshot, etc.).
- **Type of Report:** Indication about how the detected changes are reported and/or highlighted to the user.

- **Empirical Evaluation:** Insight into how the evaluation of the technique or algorithm was conducted.
- **Open Source Availability:** Indication of whether the technique or algorithm is available as open source. Nonetheless, this is complementary and not considered for the exclusion criteria.

6.1.3 Data results

The predominant area of research in detecting delta GUI changes, accounting for 59% (13 out of 22) of total papers, is aligned with Change Detection and Notification (**CDN**) systems or tools for monitoring web applications. These systems initiate a crawling process on a set of web pages specified by the user. Subsequently, a comparison method or algorithm is applied to identify the values of diverse properties associated with Document Object Model (DOM) elements that have changed. Alternatively, these systems may use visual comparison techniques to compare the web states for detecting changes. Finally, the identified changes are notified to users and exposed through reports that incorporate textual and visual information.

Research on **regression** testing, with a focus on automatic detection and reporting of delta GUI changes, accounts for 32% (7 out of 22) of total papers and encompasses diverse approaches. First, web regression testing studies from Raina et al. [231] and Walsh et al. [285] employ similar DOM comparison techniques related to CDN systems. Second, regression testing studies that delve into the automatic inference of event graphs for web systems (Roest et al., [243]) or desktop systems (Gao et al., [95]) emphasize the reuse of event graphs for different system versions to identify state changes. Third, another research strand consisting of multiple studies (Tanno et al., [269]; Adachi et al., [1]) introduces image-based methods, potentially integrable with other GUI testing tools that compare screenshots of application states between two versions to detect differences. Fourth, a study focusing on Android systems (Xiong et al., [300]) employs random GUI testing on an application version, repeating the same actions on the second version to detect widget inconsistencies.

Lastly, **other** papers account for 9% (2 out of 22) and describe tools with the capability of reporting GUI delta changes. TAO [153] is a GUI testing toolset that, in addition to including an automatic test generator and static binary analysis, includes a UI-Diff tool to track GUI changes on desktop applications. GCAT [184] is a tool with the objective of detecting and summarizing delta GUI changes during the evolution of mobile apps.

The scientific publications resulting from the systematic mapping of the literature are shown in Table 6.1.

Table 6.1 GUI Change Detection research papers obtained from the systematic mapping of the literature

Year	Title	SUT	Group	Summary
2001	Monitoring Web information changes [88]	Web	CDN	A system called CDWeb that allows users to monitor a whole web document or specific portions of their interest.
2001	WebSCAN: Discovering and Notifying Important Changes of Web Sites [229]	Web	CDN	A system called WebSCAN that monitors and analyzes the change of pre-registered Web sites and notifies important changes to users.
2001	Perception of content, structure, and presentation changes in Web-based hypertext [91]	Web	CDN	A study about how users perceive web changes and which changes they consider relevant. Then, the observations of the study were used to guide the design and development of Walden's Paths Path Manager tool.
2001	Managing change on the web [90]	Web	CDN	A detailed description of the Walden's Paths Path Manager tool used to assist maintainers in discovering when relevant changes occur to linked web resources.
2002	Information Monitoring on the Web: A Scalable Solution [159]	Web	CDN	A tool called WebCQ designed to discover and detect changes on web pages and notify users about interesting changes with personalized messages.

2004	The eShopmonitor: A comprehensive data extraction tool for monitoring Web sites [4]	Web	CDN	The eShopmonitor is a tool that allows users to monitor data of interest that has changed on commercial web-sites.
2004	Managing distributed collections: Evaluating Web page changes, movement, and replacement [71]	Web	CDN	The Walden's Paths Path Manager is a tool that allows users to monitor whether any page in a collection of web pages has changed.
2005	CX-DIFF: a change detection algorithm for XML content and change visualization for WebVigiL [126]	Web	CDN	A system called WebVigil that allows users to specify, manage, receive notifications, and view customized web page changes.
2009	TAO project: An intuitive application UI test toolset [153]	Desktop	Other	A GUI testing toolset called TAO that contains a UI-diff tool that allows to automatically track GUI changes.
2009	Changing how people view changes on the web [270]	Web	CDN	An Internet Explorer browser plugin that compares the previous cached page with the current page to highlight the way a page has changed when the user returns to it.
2009	Browsing Assistant for Changing Pages [129]	Web	CDN	A framework that provides continuous assistance to users browsing the Web regarding its temporal context.
2010	A novel approach for web page change detection system [138]	Web	CDN	A system that compares old and modified web pages to find and highlight the changes to the users.

2010	Vi-DIFF: Understanding Web Pages Changes [217]	Web	CDN	An approach called Vi-DIFF that detects content and structural changes in the visual representation of web pages.
2010	Regression Testing Ajax Applications: Coping with Dynamism [243]	Web	Regression	Crawljax is a tool that can automatically infer web state-flow graphs. Re-using information from previous web version graphs allows performing regression testing to view the added and removed states in each crawl session.
2013	An automated tool for regression testing in web applications [231]	Web	Regression	An automated tool for regression testing that can identify and report the changes in web applications.
2015	Pushing the limits on automation in GUI regression testing [95]	Desktop	Regression	GUITAR is a tool that can automatically infer event flow graphs. Re-using information from previous SUT version graphs allows performing regression testing to report widget and state mismatches.
2018	Detecting and summarizing GUI changes in evolving mobile apps [184]	Mobile	Other	A tool called GCAT for detecting and summarizing GUI changes during the evolution of mobile apps.
2020	Region-based detection of essential differences in image-based visual regression testing [269]	Image based	Regression	A visual regression testing method called ReBDiff that detects differences in the state images of two versions of an application.

2020	A Method to Mask Dynamic Content Areas Based on Positional Relationship of Screen Elements for Visual Regression Testing [1]	Image based	Regression	A visual regression testing method that allows the masking of dynamic state content when detecting differences in the state images of two versions of an application.
2020	Automatically identifying potential regressions in the layout of responsive web pages [285]	Web	Regression	A tool called REDECHECK that extracts the responsive layout of two versions of a web page and compares them, alerting developers to the differences in layout that they may wish to investigate further.
2021	WebEvo: taming web application evolution via detecting semantic structure changes [253]	Web	CDN	A tool called WebEvo for monitoring web element changes considering text and image content.
2023	An empirical study of functional bugs in Android apps [300]	Mobile	Regression	A tool called RegDroid that generates random GUI tests on two app versions and checks whether the GUI states of versions A and B contain similar widgets. If not, the inconsistency is reported as a bug.

6.1.4 Other interesting related work topics

Although certain studies were excluded due to their lack of specific focus on reporting textual or visual GUI changes in delta versions, it is relevant to acknowledge and discuss research that employs GUI change detection techniques for diverse and important objectives within software testing:

Repair test scripts

Within the realm of research dedicated to detecting GUI changes to automatically repair or provide information to repair scripts, certain studies offer direct relevance. Determining its inclusion as directly or indirectly related work is a nuanced distinction.

Grechanik et al. [101] research combines a GUI diff-tree tool with a script analyzer tool. This combination aims to generate informative messages for test engineers, aiding them in the maintenance and evolution of GUI-directed test scripts that may break when GUIs are modified between successive releases of GUI-based applications.

Zhang et al. [306] introduce FlowFixer, a tool for Java Swing desktop applications. FlowFixer instruments the old application version by recording user actions and instruments the new application version by executing random UI actions. The objective is to identify instrumented method matches and, consequently, automatically repair broken workflows in evolving GUI applications. Gao et al. [94] evaluate the SITAR technique with Java desktop applications. SITAR utilizes GUI ripping to obtain an event flow graph that represents the GUI event interactions of a new application version. In cases where a test script can not complete a path of events for execution, SITAR leverages information from the event flow graph to calculate multiple alternative repairing paths. Human testers are then notified about these alternatives to manually confirm how to repair broken scripts.

In the realm of web systems, several studies delve into strategies for repairing test scripts between two versions of websites. Choudhary et al. [61] introduce the tool WATER, which compares the behavior of test cases across successive releases of a web application to suggest repairs for broken tests. Hammoudi et al. [110] present WATERFALL, an approach that enhances the effectiveness of WATER by employing a fine-grained approach applied to the iterative versions/commits of web applications. Stocco et al. [264] propose a test repair technique that employs visual analysis implemented in a tool named VISTA. Kirinuki et al. [140] introduce the COLOR approach, which utilizes various web properties to support repairing broken locators in test scripts. Nass et al. [197] present the Similo approach, leveraging information from multiple web element locator parameters to identify the web element with the highest similarity.

In the domain of mobile systems, various studies explore methods to repair test scripts. Li et al. [154] introduce the ATOM approach, incorporating semi-automatic mechanisms to calculate delta event sequence models. These models capture changes introduced by new application versions, facilitating the maintenance of GUI test scripts. Song et al. [262] present an XPath-based approach that enables the repair and reuse of test scripts for subsequent app versions, particularly when alterations occur in the locations, names, or property values of UI controls. Chang et al. [56] propose the CHATEM approach, which involves the

semi-automatic construction of event sequence models of evolving mobile apps to maintain and generate new test scripts. Pan et al. [207] develop the METER approach, leveraging computer vision techniques to execute test script actions in the next application version. This process allows the construction of the replacement of test actions to repair broken scripts. Extending the METER approach, Xu et al. [302] present GUIDER, which incorporates structural information of app GUIs to enhance the effectiveness of repairing test scripts.

Cross-Browser and Cross-Device testing

While studies focused on testing GUI cross-browser compatibility for web applications or GUI cross-device compatibility for mobile systems do not directly employ GUI change detection for delta versions, it is pertinent to mention certain tools and techniques that offer valuable insights for their potential usage.

Mesbah et al. [179] introduce an automated cross-browser compatibility testing approach in a tool called CrossT, an extension of the web crawler Crawljax. Initially, the web application is crawled in each desired browser to infer a navigation model. Subsequently, a pair-wise comparison is performed on the generated models to report discrepancies. Tanaka et al. [267, 268] present the web application compatibility testing tool X-Brot, encompassing both functional and visual compatibility testing. The functional approach examines whether the same UI action on a web page results in a compatible action on each browser. Simultaneously, the visual technique checks whether the same web page displays a similar view on each browser. Ren et al. [235] propose CdDiff, an image-based method that helps to visualize the results of mobile application compatibility testing for different devices, operative system versions, or resolutions.

6.1.5 Actionable insights obtained from the systematic mapping of the literature

The systematic mapping of the literature process, conducted to answer Delta-RQ1, has yielded insights leading to conclusions about the research fields covered by GUI change detection techniques and their mode of dissemination:

1. Software testing change detection techniques for desktop, web, and mobile systems are frequently applied for purposes beyond specifically highlighting GUI delta changes.
2. The terminology used to describe GUI change detection approaches is diverse across existing software testing studies, with varying adoption trends over the years.

3. Research papers often lack explicit information about whether or how detected GUI changes are reported to users, such as script repair suggestions, bug change reports, or event flow graph models.
4. Studies with ambiguous abstracts, inadequate summarization of content, or non-uniform terminology necessitate an in-depth snowball process for discovery.
5. This field of research requires standardization by cataloging the objectives, terminology, practices, and challenges of delta GUI change detection.

The analysis of existing studies has allowed us to identify which technical implementations are valuable in preparing a change detection process. At the same time, it has helped us to uncover gaps in the current state-of-the-art, outlining the steps that require research and implementation efforts:

1. State models or event flow graphs serve as conceptual artifacts to represent application state-action transitions. These models are valuable for implementing system-independent GUI change detection techniques. Additionally, they facilitate the visualization of action transitions, not solely focusing on states. This information can help developers and testers visualize the pathways leading to the changed GUI states.
2. The report should be easy to understand for various users, as well as help visually by highlighting changes without using a large number of colors in an intrusive way.
3. There are image comparison techniques that can potentially be used on diverse GUI systems. However, to the best of our knowledge, there is no proposal has been evaluated on GUI state-action transitions in three different systems, such as desktop, web, and mobile applications.

6.2 Delta GUI change detection approach

As detailed in Chapter 2 of this thesis, TESTAR employs various APIs and automation frameworks to connect and interact with different types of systems. Windows Automation API and Java access bridge are used for desktop applications [189], Selenium WebDriver for web pages [240], Appium for mobile applications [128], and external plugins such as iv4XR can be used for eXtended Reality (XR) systems [226, 238]. Additionally, as explained in Chapter 3.1 and Chapter 4.1, TESTAR can infer a state model while dealing with dynamism, non-determinism, and state space explosion challenges by configuring abstraction strategies.

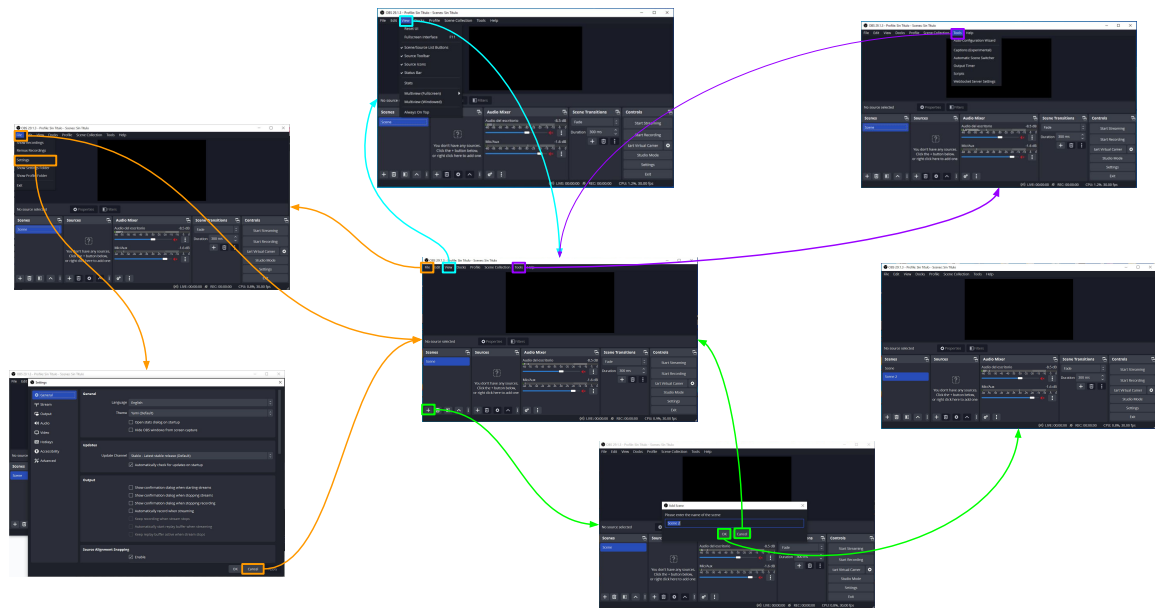


Fig. 6.4 Representation of a GUI state model of a desktop application that contains 7 states and 11 action transitions

Figure 6.4 exemplifies a GUI state model inferred by TESTAR that contains 7 discovered states and 11 action transitions that connect them.

To establish a delta GUI change detection approach, a novel tool that compares two GUI state models and highlights the changes between them has been developed. This GUI change detection tool uses the TESTAR inferred GUI state models. However, while TESTAR supports configuring abstraction strategies, additional configuration was necessary to research the state model inference strategy, particularly for inferring *partially complete models*.

6.2.1 State model inference strategy

When inferring a GUI state model, the idealistic goal could be to define an abstraction strategy capable of adequately identifying all the states and actions within a system, representing the complete logic of transitions for a SUT. Nonetheless, achieving this goal proves unrealistic when applying GUI inference techniques to real systems. The extensive number of states and actions in large and complex systems and their intrinsic dynamic behavior pose a significant obstacle to inferring a complete model. It is essential to design an *inference strategy* that aligns with a manageable objective for state model inference.

In the delta GUI change detection approach, it is necessary to infer various GUI state models for each delta version of the same application for subsequent GUI change detection analysis. Nevertheless, attempting to infer a complete model for a large and complex system is

neither efficient nor realistic due to the essentials of the state explosion challenge [195]. Any technical mitigation or advanced solution that requires implementing an abstraction technique [288] or a test-driven strategy [228] may lack the inference of part of the SUT functionality. Therefore, there is a need to find a solution that aims to balance the completeness of the inferred model while dealing with large models.

To accomplish the delta GUI change detection goal, a model is required to encapsulate all existing state-action transition functionalities. If some of these transitions are not inferred in the model, the detection of changes will provide false positives and negatives due to the lack of modeled GUI information. However, due to large and complex systems having the essential state explosion challenge, the search space of the model must be restricted to a manageable size. To achieve this objective, a possible solution is to infer a *partially complete model*. This partially complete model entails a limited set of actions with a restricted depth of state-action transitions.

For instance, Figure 6.5 represents how the inference process of the model is restricted to a limited depth, represented as the encapsulated state-action transitions in the center of the image. The external states outside the partially complete model (i.e., the greyed state and action transitions outside the central encapsulated model) are also functional parts of the SUT that may require a high-depth exploration to be discovered and inferred in the model. The main objective of the inference strategy is to allow users to customize this limited set of actions with a restricted depth depending on the size and complexity of the model being created from the SUT.

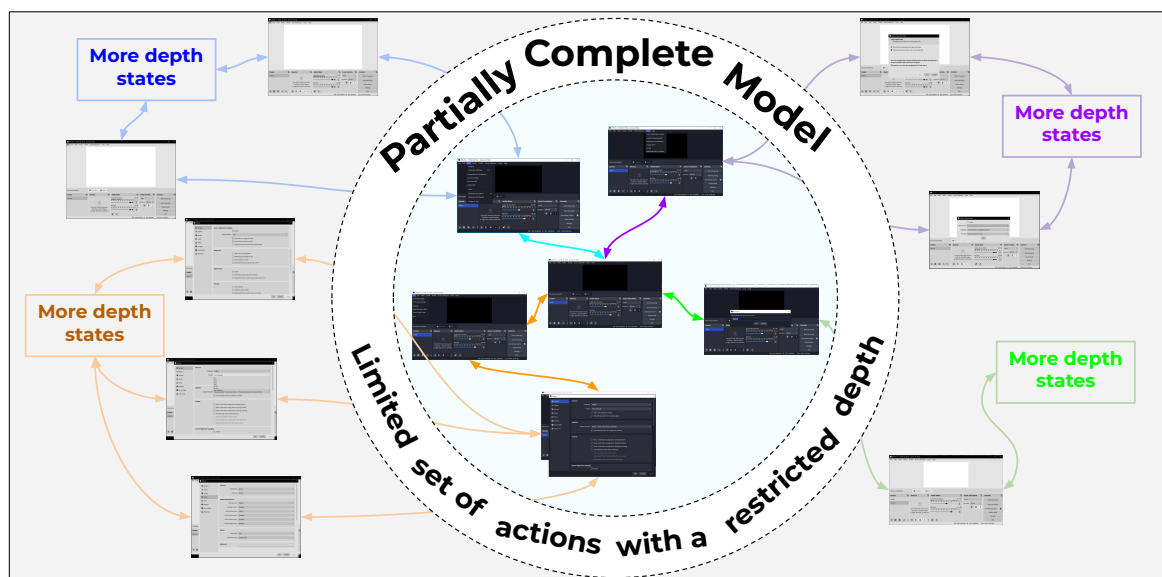


Fig. 6.5 Partially complete model inferred obtained of a desktop application due to a limited set of actions with a restricted depth

6.2.2 GUI Change Detection tool

The ChangeDetection tool ¹ compares two inferred GUI state models from distinct software versions to detect and highlight GUI changes [239]. At first, a change detection algorithm simultaneously transits the corresponding states and actions of both state models to detect and mark the states that have been changed, added, or removed. Subsequently, a merged graph technique is employed to visualize the changed states, as well as the added and removed transitions. Figure 6.6 shows two partial state models, SM_{new} (v30.0.2) and SM_{old} (v29.1.3), of two different versions of the OBS open-source desktop application ².

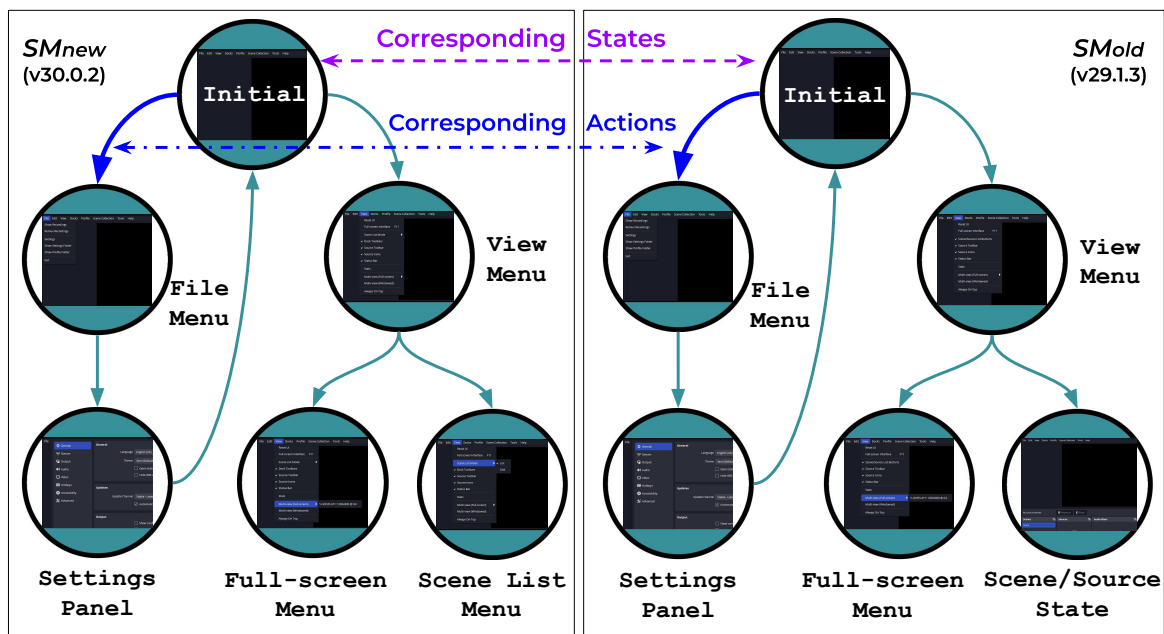


Fig. 6.6 OBS partial inferred state model from SM_{new} (v30.0.2) and SM_{old} (v29.1.3)

Change detection algorithm

The underlying idea of the **Change_Detection_Algorithm** is to recursively traverse transitions $s \rightarrow a \rightarrow s'$ in both SM_{new} and SM_{old} models while comparing the properties of the target states s' . As the identification of states and actions in the model relies on the chosen abstraction, the initial step in the algorithm validates that the models to be compared utilize the same abstraction properties (line 1). It is then that the algorithm finds the initial states of the models, establishes the initial corresponding state associations for the *newInitialState*

¹<https://github.com/TESTARtool/ChangeDetection.NET>

²<https://github.com/obsproject/obs-studio>

and *oldInitialState* (line 2), and starts to recursively transit the SM_{new} and SM_{old} models by invoking the `CompareStates` procedure (line 3).

The `CompareStates` procedure takes the corresponding *newState* and *oldState* as parameters for comparison (line 4). To prevent infinite recursion during model comparison, both corresponding states are marked as `Handled` (line 5). Then, the algorithm compares the state abstract identifiers of the corresponding states to detect whether the state has changed between versions (line 6). To continue traversing the model, the algorithm extracts a *newActionsList* of unhandled actions for *newState* (line 7) and a *oldActionsList* of unhandled actions for *oldState* (line 8). For each *newAction* in the *newActionsList* (line 9), the `CompareActions` procedure is invoked (line 10).

The `CompareActions` procedure takes a *newAction* from *newState* and the *oldActionsList* of the corresponding *oldState* as parameters (line 13). Similar to the `CompareStates` procedure, the *newAction* is marked as `Handled` to prevent infinite model recursion (line 14). First, the algorithm checks if the *newAction* has a corresponding *correspondingOldAction* in the *oldActionsList* (line 15). If *correspondingOldAction* is null (line 16), indicating the absence of a corresponding action in SM_{old} , the algorithm marks *newAction* as a new model transition and finishes the recursive comparison in this part of the model (line 17). Conversely, if a *correspondingOldAction* is found, the algorithm marks *newAction* as a matched transition (line 19) and marks *correspondingOldAction* as `Handled` (line 20). To continue traversing the model, the algorithm retrieves the *newTargetState* of the *newAction* (line 21) and the *oldTargetState* of the *correspondingOldAction* (line 22). Finally, if the subsequent *newTargetState* and *oldTargetState* have not been handled yet (line 23), the `CompareStates` procedure is invoked to continue the recursive model comparison (line 24). If the subsequent states have already been compared and handled, the recursive comparison concludes in this part of the model.

The `Change_Detection_Algorithm` traversed the models, comparing the identifiers of corresponding states and actions and marking them as matched or new. However, certain states and actions from both SM_{new} and SM_{old} models may remain uncomparing due to the difference in the abstract identifiers within the model paths. In the subsequent merge graph technique, the remaining states and actions from SM_{new} that were not compared are considered as *new*, while those from SM_{old} that were not compared are designated as *removed*.

Algorithm Change_Detection_Algorithm

```

Require:  $SM_{new}$  ▷ The inferred state model from the new application version
Require:  $SM_{old}$  ▷ The inferred state model from the old application version
1:  $CheckAbstractAttributes(SM_{new}, SM_{old})$  ▷ State models must use the same abstract properties
2:  $(newInitialState, oldInitialState) = FindInitialStates(SM_{new}, SM_{old})$  ▷ Initialize corresponding states
3:  $CompareStates(newInitialState, oldInitialState)$  ▷ Invoke the state comparison procedure
4: procedure COMPARESTATES( $newState, oldState$ )
5:    $MarkHandledStates(newState, oldState)$  ▷ Mark states as handled
6:    $CompareStateIdentifiers(newState, oldState)$  ▷ Compare state identifiers to detect changes
7:    $newActionsList = FindUnhandledActions(newState)$  ▷ Retrieve the actions for the new state
8:    $oldActionsList = FindUnhandledActions(oldState)$  ▷ Retrieve the actions for the old state
9:   for  $newAction \in newActionsList$  do ▷ Iterate through each new action to
10:     $CompareActions(newAction, oldActionsList)$  ▷ Invoke the actions comparison procedure
11:   end for
12: end procedure
13: procedure COMPAREACTIONS( $newAction, oldActionsList$ )
14:    $MarkHandledAction(newAction)$  ▷ Mark the new action as handled
15:    $correspondingOldAction = FindCorrespondingAction(newAction, oldActionsList)$ 
16:   if  $correspondingOldAction$  is NULL then ▷ If there is not a corresponding action in the old model
17:      $SetTransitionAsNew(newAction)$  ▷ The new action is a new transition
18:   else ▷ If a corresponding action exists in the old model
19:      $SetTransitionAsMatch(newAction)$  ▷ The new action is an existing matched transition
20:      $MarkHandledAction(correspondingOldAction)$  ▷ Mark the corresponding old action as handled
21:      $newTargetState = GetState(newAction)$  ▷ Get target state of transitioned new action
22:      $oldTargetState = GetState(correspondingOldAction)$  ▷ Get target state of transitioned old action
23:     if  $NotHandledStates(newTargetState, oldTargetState)$  then ▷ If those states were not handled
24:        $CompareStates(newTargetState, oldTargetState)$  ▷ Invoke next states comparison procedure
25:     end if
26:   end if
27: end procedure

```

Merge graph technique

The merge graph technique allows two graphs to be merged into one graph for further visualization and analysis [16]. After the ChangeDetection algorithm has transitioned the corresponding states and actions of both SM_{new} and SM_{old} models, this technique creates a merged graph model SM_{merged} and executes two merging steps that can be visualized in Figure 6.7:

1. Add all states and actions from SM_{new} .

The model SM_{new} contains three possible categories of states after the execution of the ChangeDetection algorithm.

First, the corresponding states remain identical in both delta versions (e.g., Initial, File Menu, and Settings Panel). These identical states are visualized with opaque circles.

Second, the corresponding states contain changes between delta versions (e.g., View Menu). These changed states are visualized with a dashed linear border.

Third, there are state-transitions that existed in SM_{new} and in which the ChangeDetection algorithm has not found a corresponding state in SM_{old} (e.g., Full-screen Menu - new and Scene List Menu). These are newly added state-transitions visualized with a green star.

2. Add non-matching states from SM_{old} and wire actions.

The state-transitions that were not handled in SM_{old} during the ChangeDetection algorithm are state-transitions that do not exist in SM_{new} . Hence, these are removed state-transitions between delta versions (e.g., Full-screen Menu - old and Scene/Source State). These are old removed state-transitions visualized with a red triangle.

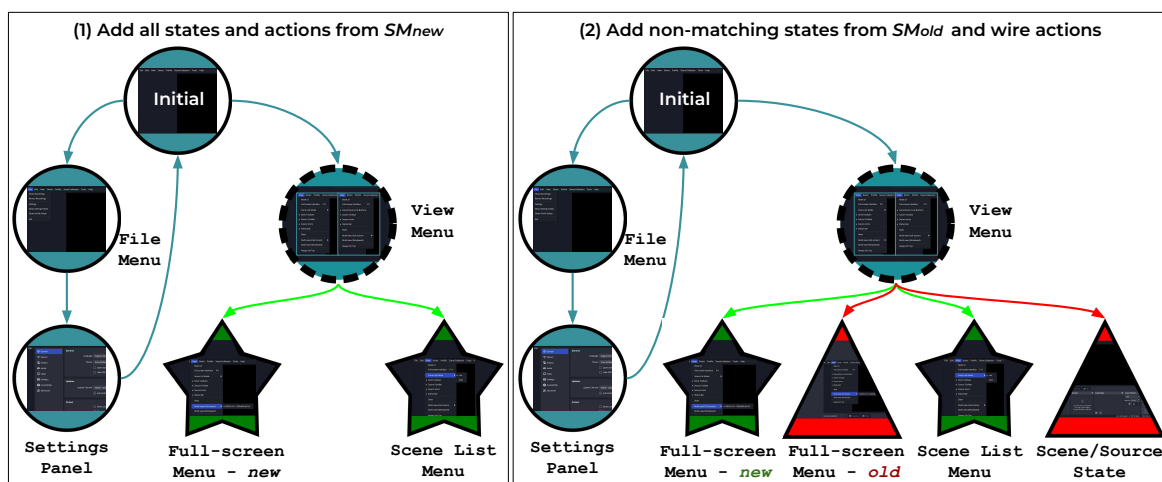


Fig. 6.7 Merge graph technique for visual graph comparison with OBS partial inferred state models

The SM_{merged} serves as an interactive model, enabling users to choose specific states and actions for visualization and analysis of change detection results. Green stars and red triangles show screenshots of the newly added or old removed states, respectively. Circle states, which indicate identical abstract states, display two screenshots—one for the new state and one for the old state. Finally, changed states represented by dashed linear borders, in addition to the two screenshots of the new and old states, generate a third screenshot created using a pixelmatch library³ to highlight the GUI changes (see Figure 6.8).

³<https://github.com/mapbox/pixelmatch>

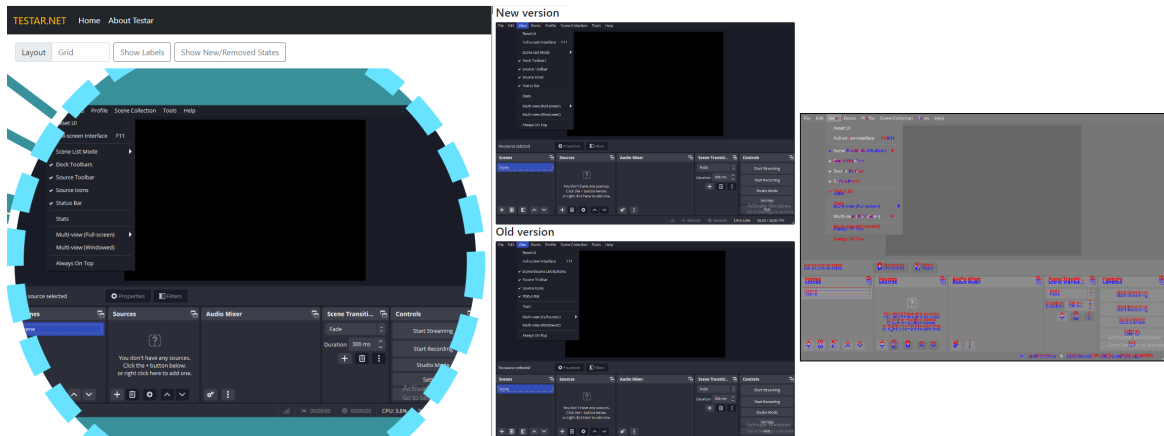


Fig. 6.8 Visual representation of changed states

Configurable action abstraction identifiers

In the merged model from Figure 6.9-A, the transition to the Full-screen Menu state is detected at the same time as added and removed state-transitions. This is provoked because the abstraction strategy for identifying actions relies on the abstract identifier of the origin View Menu state, which has undergone changes.

In our prior research [239], we observed that this behavior may not only result in a visually confusing representation for users but can also impede the effectiveness of the ChangeDetection approach. If the corresponding state associations disappear due to a state-transition originating from a changed state, all subsequent transitions from the Full-screen Menu state will be incorrectly regarded as entirely newly added or removed.

State models can employ descriptions to identify action transitions $s \rightarrow a \rightarrow s'$ (e.g., *Left click at 'Click Full-screen'*). This action description can remain consistent even if the abstract identifiers of the origin states have changed between versions. For this reason, we opted to extend the ChangeDetection tool with a configurable option that enables users to decide if the tool needs to use the action abstract identifier or the action description.

Figure 6.9-B illustrates this configurable alternative for the same merged model. In this case, the tool detects that the action description of *Left click at 'Click Full-screen'* remains consistent across both OBS versions. As a result, the Full-screen Menu is marked as a changed state. This involves considering the Full-screen Menu state as corresponding states in the traverse algorithm, potentially enabling subsequent transitions to be compared.

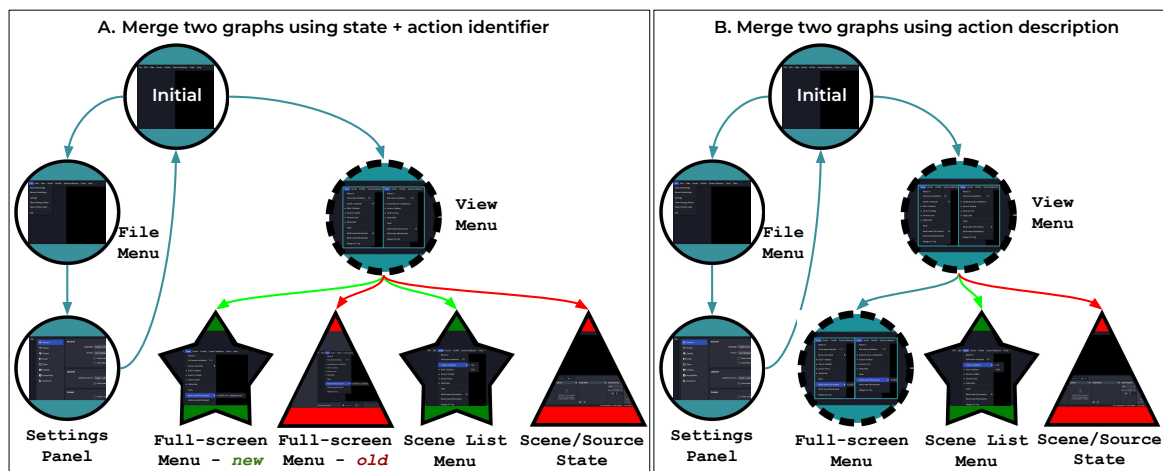


Fig. 6.9 Merge graph technique comparison using state and action abstract identifier vs. action description

6.2.3 Empirical Study

The objective of this study is to evaluate the importance of using a GUI change detection approach to facilitate the identification and visualization of delta GUI changes. This will help to validate that the GUI changes have occurred as intended or to reveal unforeseen GUI modifications. To achieve this, we intend to verify whether the automated utilization of state model inference through a scriptless tool, followed by the application of the ChangeDetection tool, effectively highlights changed, added, and removed states in delta changes of software projects.

In order to guide the study, we have formulated the following research question:

- Delta-RQ2: Does the ChangeDetection tool detect delta GUI changes when using inferred state models?

To answer Delta-RQ2, we designed a controlled experiment following the guidelines proposed by Wohlin et al. [297] and aligned with a methodological framework specifically built to evaluate software testing techniques [278]. For the purpose of validating the tool with diverse application systems and disseminating the results, we have opted to select one desktop, one web, and one mobile open-source system.

This experiment entails the inference of multiple GUI state models from iterative versions of the same system, followed by the application of the ChangeDetection tool to each pair of model versions within the same system. For each ChangeDetection comparison between version pairs, we quantify and verify the number of detected GUI changes.

To analyze the change detection results and be able to answer the research questions, we defined the following null hypothesis:

- H_0 : The ChangeDetection tool does not detect delta GUI changes using inferred state models.

SUT Objects

The SUTs selected for this experiment, suitable for the change detection approach, must be published under an open-source license to allow software analysis and facilitate the replication of the experiments. Moreover, the TESTAR tool should be able to connect and detect the GUI widgets in order to infer a state model. Based on these open-source requirements, we selected the following SUTs:

- OBS Studio [203] is an open-source desktop application designed for capturing, recording, and streaming video content. At the end of 2023, it is the 6th most-starred C code software project on GitHub. It is widely known by various categories of users (e.g., teachers, event organizers, streamers, etc.) and is actively under development and maintenance.
- Calibre-Web [49] is an open-source web application that offers a clean and intuitive interface for browsing, reading, and downloading eBooks. It is also a widely known application and is actively under development and maintenance.
- MyExpenses [273] is an open-source mobile Android application designed to keep track of user expenses and incomes and to export them in different file formats. Even though it is the least GitHub-starred of the three systems, the application has been in active development and maintenance for more than 10 years.

The details obtained at the end of 2023 regarding the selected SUTs are presented in Table 6.2. The lines of code (LLOC), together with the GitHub stars, issues, and contributors, indicate that these are not toy software projects, i.e., they are representative SUTs of real and widely adopted applications. For each SUT, we selected 4 recent versions to evaluate the change detection approach. We consider the first selected version the control baseline, and the following versions will be used to evaluate the detect GUI delta changes [297].

The TESTAR tool can be employed for testing purposes through the implementation of diverse oracles that verify the presence of failures in the system's GUI. However, given that this study does not focus on the validation of the scriptless technique for fault detection, we have opted to apply the blocking principle [297] to disable TESTAR oracles.

Table 6.2 Details of the selected SUT Objects

Metric	OBS-Studio	Calibre-Web	MyExpenses
SUT type	Desktop	Web	Android
GitHub Stars	52,700	10,600	630
GitHub Issues	3,300	2,300	1,100
Contributors	575	198	56
LOC	647,352	270,920	249,021
SUT Versions	v27.2.4 (Mar 30, 2022) v28.1.2 (Nov 5, 2022) v29.1.3 (Jun 19, 2023) v30.0.2 (Dec 10, 2023)	v0.6.18 (Apr 3, 2022) v0.6.19 (Jul 31, 2022) v0.6.20 (Mar 27, 2023) v0.6.21 (Oct 21, 2023)	v3.6.6 (Oct 30, 2023) v3.6.7 (Nov 13, 2023) v3.6.8 (Nov 21, 2023) v3.6.9 (Dec 4, 2023)

Independent variables: Inference strategy

The objective of this study is to validate if the ChangeDetection tool is capable of detecting and reporting changes between SUT versions. To accomplish this objective, it is necessary to infer a partially but complete state model that represents all the existing GUI transitions of a section of the SUT. If the inference process is not restricted and controlled, the obtained state model may result too big and complex, potentially inducing a wide variety of dynamism, non-determinism, and state explosion challenges. For this reason, we defined a set of independent variables regarding the inference depth limit and the actions to derive and execute in the inference process.

First, we designed a similar **sequences inference strategy** (see Table 6.3) based on re-launching the SUT Objects multiple times and limiting the maximum depth of the models. Running a large number of sequences of 3 action lengths reduces the number of discovered states but facilitates the inference of the partially but complete state model. Additionally, it mitigates abstraction challenges that may arise and makes it easier for users to define an appropriate abstraction strategy.

Another important inference factor to consider is the time duration given to the actions to execute the GUI-events interactions and the time to wait between the execution of actions. The action duration of 0 seconds, indicates the TESTAR tool that it is mandatory to teleport the mouse to the interactable widget coordinates and perform the GUI-event interaction. This is important in the inference process of desktop and web GUI systems since mouse movements instead of mouse teleport can trigger mouse-over GUI events and provoke unintentional GUI alterations. Finally, because applications require time to load widgets completely in the GUI, we decided to give all of them 2 seconds after each action execution.

Second, we configured an **action derivation strategy** (see Table 6.4) aiming to control the execution of actions and potentially reduce the abstraction inference challenges:

Table 6.3 Independent variables for sequences inference strategy

	Depth limit	Action duration	Wait after action
OBS-Studio	3 actions	0 second	2 second
Calibre-Web	3 actions	0 second	2 second
MyExpenses	3 actions	0 second	2 second

- *Force* actions are specific step-by-step actions necessary to start the SUT in the initial desired state.
- *Click* actions indicate with which type of widgets we focus on click-interacting in the model inference process.
- *Filter* actions are used to ignore widgets that open external system processes, ignore dynamic or high-combinatorial widgets that increase the state model size, and ignore detected widgets that create non-deterministic transitions.
- *Kill* actions are intended to detect and close web browser and file explorer processes that appear during the inference process. These are implemented explicitly for the OBS desktop SUT because it is a widely used functionality in various widgets, and defining rules to filter all these widgets would require a lot of effort and maintenance over versions.

The *Force* actions executed to start the model in the initial state (e.g., login, close update initial panel) are not inferred in the model and are not considered part of the maximum depth limit. However, this is an inference disadvantage in this study, since states forced before reaching the desired initial state are not tracked in the model for subsequent change detection.

Typing actions are essential to interact with the SUT and manage OBS scenes, Calibre books, or money Expenses. Nevertheless, since the inference strategy focused on clicking to discover existing menus, lists, and configuration panels already provide sufficient states and actions in the models to perform the change detection evaluation, we decided not to derive typing actions.

Sliding actions are challenging for the state model inference process since they change the visible and interactable widgets in the GUI. This affects the abstract strategy and increases the complexity of the exploration space. For this reason, we decided not to derive sliding actions.

Table 6.4 Independent variables for action derivation strategy

OBS-Studio	<p>(Force) Close the update panel that appears in the GUI when a new version is available to be installed</p> <p>(Click) Focus on MenuItem and ListItem widgets to infer a partially complete model of the configuration menus. This also prevents interacting with dynamic widget-icons</p> <p>(Click) Close or Cancel buttons if there are no MenuItem and ListItem widgets in the state</p> <p>(Filter) Widgets that open the update version panel and widgets that, in a non-deterministic amount of seconds, verify the integrity of the files</p> <p>(Kill) Web Browser and File Explorer processes that are invoked when interacting with video and audio management widgets</p>
Calibre-Web	<p>(Force) Login with valid credentials to start in the initial GUI state as an admin user</p> <p>(Click) All the various types of clickable web widgets in this SUT (hyperlinks, buttons, checkboxes)</p> <p>(Filter) One specific List-grid widget that provoked non-determinism when transiting again to the state</p> <p>(Filter) Widgets that logout, and widgets that save or download files</p>
MyExpenses	<p>(Force) Skip the initial configuration and focus on opening the wallet, plus another action that opens the manage accounts menu</p> <p>(Click) All widgets with enabled and clickable Android attributes</p> <p>(Filter) Menu widgets in the initial state that can be shown/hidden. This provokes non-determinism and, trying to address it, a significant increase in the size of the model</p> <p>(Filter) Widgets that open states with completely dynamic date and hours information</p> <p>(Filter) Widgets with enabled and disabled switch functionality that increase the size of the model due to the combinatorial possibilities</p> <p>(Filter) Tell-a-friend widget that opens an Android system menu trying to send an email</p>

Independent variables: Abstraction strategy

The abstraction strategy consists of implementing a main abstraction mechanism and a set of sub-strategies. The main abstraction mechanism uses widget properties with the objective of distinguishing the abstract states and actions in the state model that represents the SUT behavior. Table 6.5 shows the widget properties used for OBS-Studio, Calibre-Web, and MyExpenses.

Customizing the depth limitations and action derivation in the inference strategy reduces the inference complexity and the need to deal with abstraction challenges. Even so, in

Table 6.5 Independent variables for the main abstraction mechanism

OBS-Studio	
State	Path, ControlType
Action	OriginState + OriginWidget + ActionRole
Calibre-Web	
State	WebId, WebTextContent
Action	OriginState + OriginWidget + ActionRole
MyExpenses	
State	AndroidXPath, AndroidText, AndroidClassName
Action	OriginState + OriginWidget + ActionRole

complex systems, it can be necessary to design a set of abstraction sub-strategies that complement the main abstraction mechanism to enhance the abstract states and actions identification or to deal with dynamism and non-determinism. Table 6.5 shows the following implemented types of abstraction sub-strategies:

- *Added* properties of specific widgets to the main abstraction mechanism. This is necessary if the default property values are not enough to differentiate widgets from each other.
- *Ignore* widgets that can be dynamically added or removed from the state during the inference or dynamic properties of specific widgets.
- *Event* SUT behaviors that may provoke non-determinism in the model during the inference process need to be addressed with special triggered actions.

The OBS-Studio and MyExpenses SUT objects are distinguished in their inverse abstraction strategies. In the case of OBS-Studio, we decided to omit the Title property in the main abstraction mechanism since various widgets have dynamic Title values. Then, we incorporated the Title property of MenuItem, ListItem, CheckBox, and ComboBox widgets as an abstraction sub-strategy. Conversely, in the case of MyExpenses, we decided to include the AndroidText property in the main abstraction mechanism and custom abstraction sub-strategies to ignore the dynamic widgets. Both options can be valid to implement, and they also depend on the depth limit and derived action from the inference strategy, or the complexity of the functionalities of the SUT.

Caliber-Web did not require various abstraction sub-strategies because the combination of the inference strategy and the main abstraction mechanism was adequate.

Table 6.6 Independent variables for abstraction sub-strategies

OBS-Studio	<p>(Added) The Title property for MenuItem, ListItem, CheckBox, and ComboBox widgets since these widgets are not dynamic with a depth limit of 3 actions. Moreover, we aim to detect if those Title properties have changed between SUT versions</p> <p>(Added) The identifier of the origin state on which the (Kill) Web Browser and File Explorer actions are performed</p> <p>(Ignore) All dynamic widgets from the below usage status bar and from the panels that show the usage CPU and MEM stats</p> <p>(Event) Force a teleport of the mouse to the top-left coordinates of the screen (1) at the beginning of each sequence or (2) if, after executing an action, the mouse coordinates over a widget triggers a pop-up tooltip message that remains around 10 seconds</p>
Calibre-Web	<p>(Event) Force a teleport of the mouse to the top-left coordinates of the screen at the beginning of each sequence</p>
MyExpenses	<p>(Added) The Checked property of CheckedTextView widgets to prevent non-determinism when validating a form and transit to a destination state</p> <p>(Ignore) The dynamic AndroidText property of widgets that contain a date (HH:MM AM/PM) or hours (DD/MM/YY) patterns</p> <p>(Ignore) The dynamic AndroidText property of widgets from a Calculator</p> <p>(Ignore) The dynamic AndroidText of dropdown spinner widgets when they are closed, as they can create a high combination of possibilities. These dropdown spinners are not (Filter) such as the switch widgets, because we want to detect changes in the values when they are open.</p> <p>(Event) Wait 10 seconds if, after executing an action, the state contains a temporal snackbar message widget that remains around 5 seconds. In this system, it is not possible to just move the mouse away.</p>

Effort time for independent variables

Deploying and analyzing the versions of the SUT objects to custom these independent variables requires effort to identify, configure, and validate their adequacy. The proficiency obtained from previous research in learning to design inference and abstraction strategies for desktop and web systems allowed us to reduce the effort time in this study. Principally, it helped the decision to prepare an inference strategy that restricts the depth to infer the partially complete models [239]. Table 6.7 contains an approximation of the times required to manually configure and refine the inference and abstraction strategies, as well as the automated pre-execution time it took to infer state models from different versions to validate the partially complete models where inferred adequately.

OBS-Studio required approximately 10 hours of manual analysis combined with 8 hours of automated pre-executions to obtain the appropriate configuration. Most of the effort was invested in determining the best action derivation and abstraction sub-strategy to deal with the opening of the web browser and file explorer and deciding to completely ignore the dynamic stats widgets.

Calibre-Web required approximately 6 hours of manual analysis combined with 4 hours of automated pre-executions to obtain the appropriate configuration. In the initial analysis, we recognized the robust implementation of the WebId property along the different versions of the SUT. The widget that provoked non-deterministic was the unique encounter challenge we decided to filter within the action derivation strategy.

MyExpenses required approximately 15 hours of manual analysis and 25 hours of automated model inference pre-executions to decide about an appropriate configuration. In this SUT, it was necessary to perform more preliminary experiments to find an appropriate balance with the inference and abstraction strategies to control the size of a deterministic model.

Table 6.7 Effort time to design the independent variables and run pre-executions

	OBS-Studio	Calibre-Web	MyExpenses
(Manual) Strategies configuration	10 hours	6 hours	15 hours
(Automated) Pre-executions	8 hours	4 hours	25 hours

Dependent variables

To answer Delta-RQ2:, we performed a qualitative evaluation [278]. This evaluation is manually performed by an expert in GUI testing with more than five years of experience, which aims to validate whether the detected states marked as changed indeed contain GUI changes. Subsequently, we need to check if certain SUT functionalities or displayed GUI changes do not correspond with the delta SUT version changes. Finally, it is important to examine the results obtained from these complex SUTs to refine the process for further improvements. To accomplish this, we analyze the detected delta GUI changes with the related open-source software versions. In this way, we will evaluate the following:

1. True positives change results: The state model inference and the change detection process correctly detect GUI changes that align with software delta version changes.
2. Challenging results: The state model inference and the change detection process indicate the detection of GUI changes that do not align with the expected results.

- Further improvements: The state model inference and the change detection process require improvements to detect and highlight delta GUI changes with complex SUTs.

Design of the experiments

We use the TESTAR tool with the independent variables configurations to infer a state model for each delta version of a SUT. This inference outputs 4 state models for each SUT. Then, we apply the ChangeDetection tool with the pairs of delta state models of the same SUT. This change detection comparison reports 3 merged state models with detected changes for each SUT.

Figure 6.10 shows the overview of the architecture. The inference process is similar for all SUTs, but because each system is inherently different, we explain their distinction execution below:

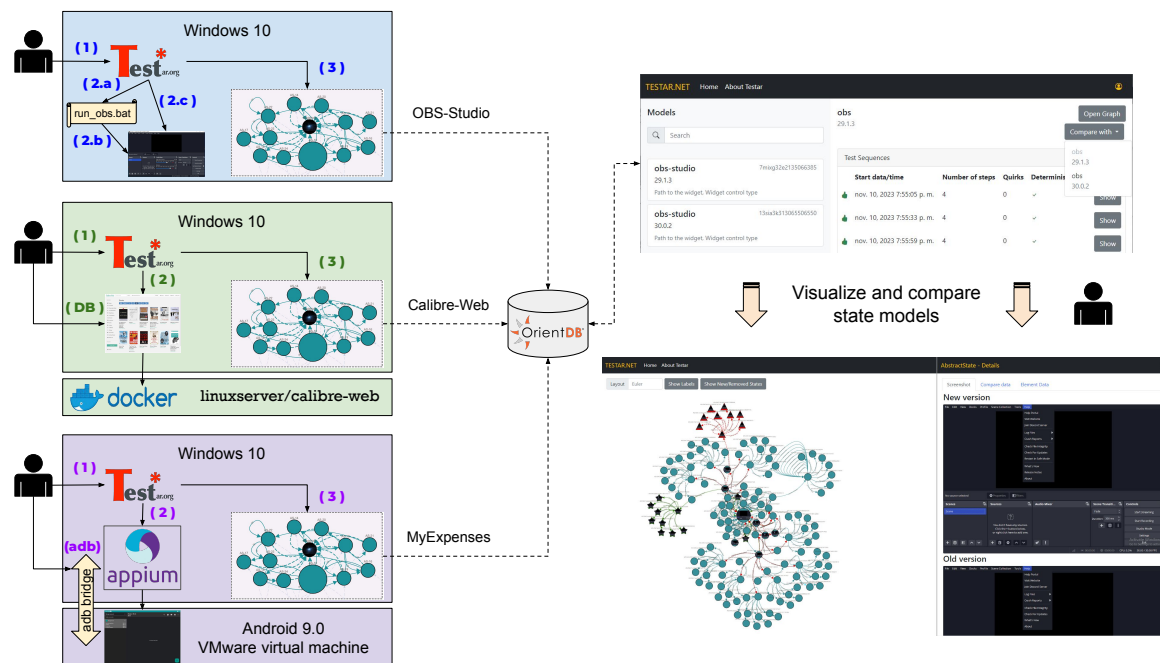


Fig. 6.10 Architecture of GUI Delta Change Detection experiments

- OBS-Studio:** This SUT is a portable Desktop application that does not require any installation process in a Windows environment. When TESTAR is launched (1), an intermediate batch script (2. a) changes to the specific OBS directory and launches the application executable (2. b). Then, TESTAR is able to connect to the OBS process (2. c) and automatically explore the SUT to infer the state model (3). The resolution in the Windows 10 machine that renders the GUI is 1200x800.

- **Calibre-Web:** This SUT is a Web application that can be deployed using a Docker version image. Before launching TESTAR, it is necessary to configure a Calibre database to store the user books. In our experimentation, we deployed the Docker image in an Ubuntu environment and manually selected the Calibre database offered in the GitHub repository (DB). Then, when indicating the web URL to TESTAR (1), the tool is able to connect to the SUT (2), and automatically explore the SUT to infer the state model (3). The resolution in the Windows 10 machine that renders the GUI is 1200x800.
- **MyExpenses:** This SUT requires an Android system to be deployed. We created an Android 9.0 virtual device in a VMware environment and connected it with the Windows 10 host that contains TESTAR by creating an adb bridge⁴. When TESTAR is launched (1), the Appium⁵ server detects the connected device and acts as middleware to obtain the state information (2). This permits TESTAR to automatically explore the SUT to infer the state model (3). The resolution of the Android 9.0 virtual device that renders the GUI is 1024x720.

At the end of the GUI state model inference processes, users can deploy the ChangeDetection tool using one Docker server image and one Docker client image. By specifying the IP and port address of OrientDB, users can inspect the state models individually and execute the comparison feature to perform the ChangeDetection algorithm. The results of the algorithm are automatically used by the merge graph technique, providing users with visualization, manual interaction, and analysis of GUI change detection results.

6.2.4 Results

This section presents the state model inference and change detection approach results. First, we summarize the inference process and size of the obtained GUI state models using the aforementioned independent variables configurations. Second, we present and discuss, for each SUT object, the results of comparing each delta version with the corresponding baseline to examine the qualitative evaluation for GUI change detection. The replication package with the state models and a complete document with visual results can be found here⁶.

⁴<https://developer.android.com/studio/command-line/adb>

⁵<https://appium.io/>

⁶<https://doi.org/10.5281/zenodo.13712379>

State Model Inference

Table 6.8 presents an overview of the inferred GUI state models, emphasizing the size in terms of discovered abstract states and executed abstract actions. Additionally, we provide information about the approximate time it took to infer the partially complete models for each SUT. The size of the models varies depending on the depth limit of the inference strategy and the properties information customized in the abstraction strategy. The inference time may vary depending on the action duration, wait after action, and force login or starting actions adopted in the inference strategy (i.e., TESTAR waits 10 seconds when starting OBS, and TESTAR takes around 10 seconds to perform the MyExpenses login). Although the model size and inference time are not directly related to the delta GUI change detection evaluation, these are essential aspects when considering techniques for evaluating rapid software delta increments.

Table 6.8 State Model Inference results

State Model	Abstract States	Abstract Actions	Inference time
OBS-Studio v27.2.4	132	319	2h 5m
OBS-Studio v28.1.2	132	265	1h 40m
OBS-Studio v29.1.3	132	198	1h 10m
OBS-Studio v30.0.2	134	206	1h 10m
Calibre-Web v0.6.18	25	436	1h 25min
Calibre-Web v0.6.19	35	499	1h 50min
Calibre-Web v0.6.20	37	499	1h 50min
Calibre-Web v0.6.21	37	499	1h 50min
MyExpenses v3.6.6	123	243	1h 50min
MyExpenses v3.6.7	125	246	1h 55min
MyExpenses v3.6.8	124	248	2h
MyExpenses v3.6.9	125	248	2h

For each SUT object, the same independent variables configuration has been used to infer the delta GUI state models. The inference and abstraction strategies for the GUI state models inference have remained resilient, requiring no adjustments to be adapted to the appearance of new dynamic widgets or those inducing non-deterministic behaviors. Nonetheless, as we mention in the following change detection results, the appearance of new widgets disrupts with *noise* the change detection technique. This *noise* is produced when widgets are either newly introduced, removed, or changed within the main static menus and panels of the SUT.

OBS-Studio results

Figure 6.11 shows an overview of how the OBS merged results are reported to users through an interactive web interface. This merged model includes distinct visual elements. Opaque green circle states signify OBS states that persist unchanged across delta versions. Dashed states that contain small images are changed states from OBS that have undergone GUI modifications. Red triangle transitions represent states and actions that have been removed in the delta software increment, while green star transitions highlight newly added states and actions. Clicking on any of these elements displays an informative panel on the right side of the web interface, presenting detailed images and textual information that users can use for the analysis and understanding of GUI changes.

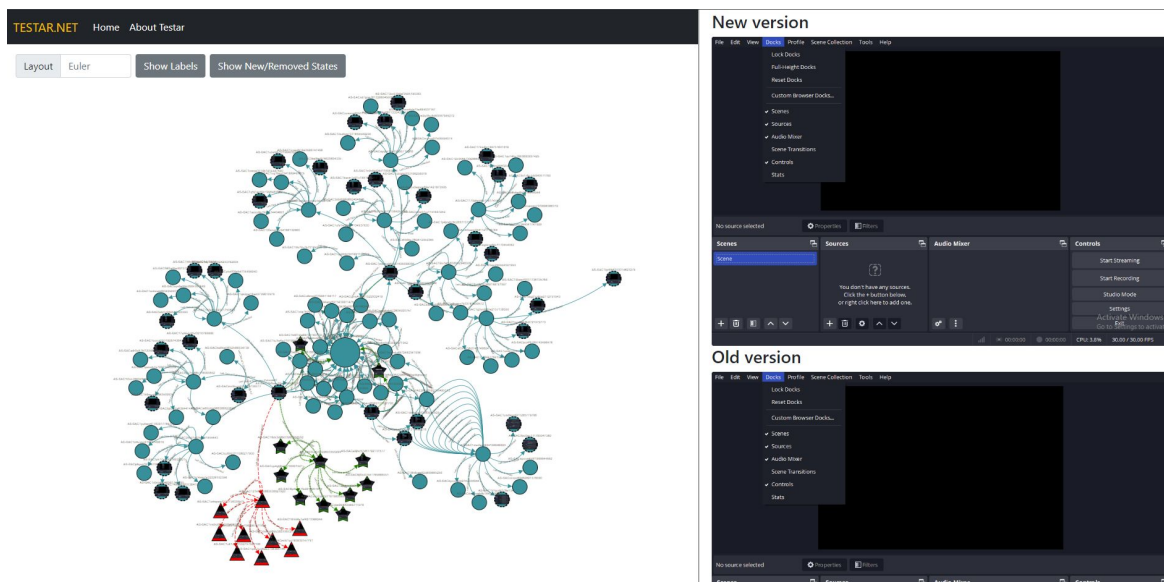


Fig. 6.11 Interactive web interface of the GUI change tool that creates a merged partially complete model from the OBS SM_{new} (v30.0.2) and SM_{old} (v29.1.3) versions with a depth of 3 actions

Table 6.9 summarizes the change detection results. The change detection algorithm for OBS-Studio has been executed using the action description. The states that have not been reached due to state model inference restrictions (action derivation and filters strategy, or depth limit) are not included in the table. Furthermore, the states that contain changes ignored by the abstraction mechanism, such as the version and contributors texts changed in the About panel, are also not included in the table.

Table 6.9 OBS-Studio change detection qualitative results

OBS-Studio	Summary of detected changes
v27.2.4 to v28.1.2	<ul style="list-style-type: none"> - ALL states: The main Audio Mixer and Scene Transitions dock panels, on the bottom side of the application, contain new icon-buttons. This provokes the change detection algorithm to detect that all states have changed. - Help menu: contains the new menu item Check File Integrity - Crash Reports menu item: contains a changed text from Upload Last Crash Report to Upload Previous Crash Report - Log File menu item: contains a changed text from Upload Last Log File to Upload Previous Log File - Add Profile panel: title has changed compared to the previous Add Scene Collection title - Tools menu: has removed Declink Captions and Declink Output menu items and added a new obs-websocket Settings item - Declink Output panel: has been removed - Declink Captions panel: has been removed - obs-websocket Settings panel: has been added - Advanced Audio Properties panel: has changed with new audio values - Transform menu item: includes new special shortcut texts for the Copy Transform and Paste Transform - File menu: has changed by removing the menu item Always on top - Docks menu: has changed the Lock UI and Reset UI menu items to Lock Docks and Reset Docks - View menu: has changed by adding the menu items to Reset UI and Always on top. Moreover, the menu items text has changed from Multiview and Fullscreen to Multi-view and Full-screen - Multi-view (Windowed) panel: has been added compared to the previously removed Multiview (Windowed) panel - Settings panel: has changed with the new option Accessibility - Settings-Hotkeys panel: has changed with a new filter by the hotkey list option Split Recording File - Settings-Output panel: has changed with a new dropdown option Encoder Preset - Settings-Advanced panel: has changed with new dropdown options SDR White Level and HDR Nominal Peak Level

v28.1.2 to v29.1.3	<ul style="list-style-type: none"> - Settings-Accessibility panel: is new in the Settings panel - ALL states: The main Scenes dock panel, on the bottom-left side of the application, contains a new Open Scene Filters icon-button. This provokes the change detection algorithm to detect that all states have changed. - Help menu: contains a new menu item What's New - Tools menu: has changed by renaming the menu item obs-websocket Settings to WebSocket Server Settings - WebSocket Server Settings panel: has been added compared to the previously removed obs-websocket Settings panel - Multi-View (Full-screen) menu item: has changed to indicate the naming of the DISPLAY. This affects the subsequent action transition - Settings-General panel: has changed due to moving the general checkbox Automatically check for updates at startup to a new Updates subpanel - Settings-Accessibility panel: has changed the checkbox text Colors to Colours - Settings-Output panel: has changed due to renaming the dropdown Encoder to Video Encoder and adding a new Audio Encoder drop-down. Moreover, has added an Audio Track option to the Recording subpanel - QComboBox Collapsed attribute has been changed in the Qt implementation. This provokes combo box items not to be displayed and clickable until the combo box is expanded
v29.1.3 to v30.0.2	<ul style="list-style-type: none"> - Docks menu: contains a new menu item Full-Height Docks - Help menu: contains two new menu items Restart in Safe Mode and Release Notes - Restart panel: has been added to the new SUT version - View menu: has changed by removing the menu item Scene/Source List Buttons and adding two new menu items Scene List Mode and Dock Toolbars. This affects the subsequent action transition - Scene List Mode menu item: has been added to the new SUT version - Settings-Hotkeys panel: has changed the hotkey list option Studio Mode into two list options Enable Studio Mode and Disable Studio Mode

- **Settings-Output panel:** has indirectly changed due to moving the settings warnings float at the bottom of the page. This change is not visual in the state but exists as a widget tree change, and it has been detected by the change detection algorithm
- **Settings-Advanced panel:** has changed by adding a new IP Family option to the Network section. This change is not visual in the state but exists as a widget tree change, and it has been detected by the change detection algorithm

Based on the evaluation of the results obtained with this complex OBS-Studio desktop application, we can consider the challenging functionalities that negatively affect the state model inference and change detection process, as well as the further improvement ideas for the analysis of the detected changes.

OBS-Studio challenging results

In the delta version from v27.2.4 to v28.1.2, the main Audio Mixer and Scene Transitions dock panels affect the change detection process of all the SUT states. Similarly, this occurs in the delta version from v28.1.2 to v29.1.3 with the new Open Scene Filters icon-button added in the Scenes dock panel.

These new widgets introduced in the main static panels do not interfere with the state model inference process itself but introduce change detection *noise* in all the states of the merged model (see Figure 6.12). This change detection *noise* implies that all states will be marked as changed. Then, the user needs to analyze all states to determine if there are changes or not.

OBS-Studio further improvements ideas based on results

Given alterations in GUI style, menus, and panel dimensions, using the pixelmatch technique proves insufficient in effectively assisting users in visualizing specific GUI changes. For instance, as illustrated in the previous Figure 6.8, the pixelmatch technique highlights changes when the GUI widget coordinates have a slight deviation due to style changes, even if the widgets are the same with identical properties. A visual improvement can consist of comparing the abstraction identifier of the widgets of the two changing states, specifically detecting discordant widgets and using their position on the screen to highlight them with a rectangle.

In the delta version from v29.1.3 to v30.0.2, the **Settings-Advanced panel** contains a new IP Family change in the widget tree that is not directly visible to the user when inspecting the merged graph. It has been necessary a complementary GUI and code user analysis to

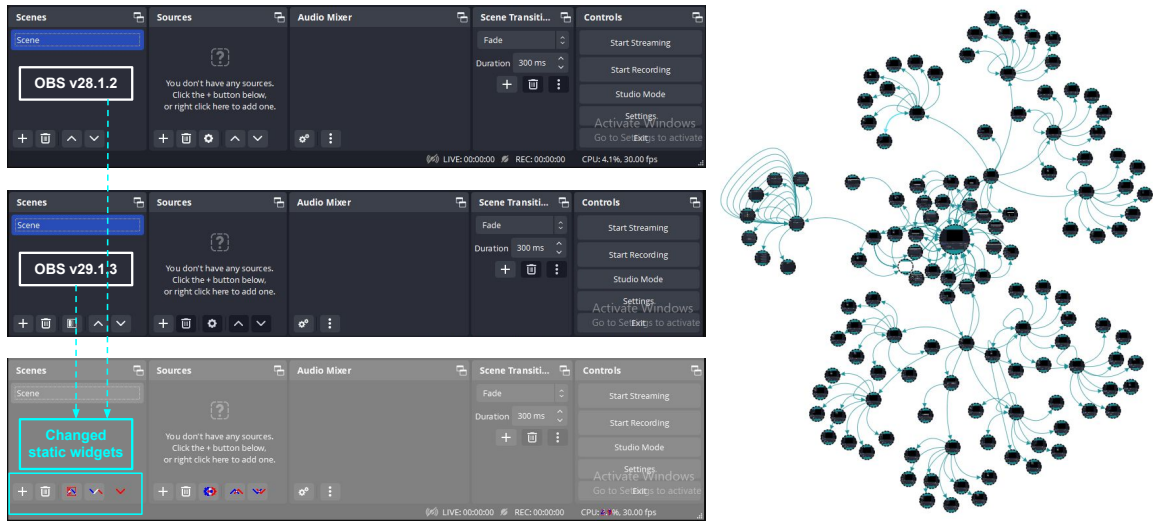


Fig. 6.12 Static widget changes challenge: The new Open Scene Filters icon-button added in the Scenes dock panel from v28.1.2 to v29.1.3 introduces *noise* in all states since they are all marked as changed

validate the existence of this delta change. A descriptive message indicating the changed widget together with the visual screenshot, could help users to recognize the changes easily.

Calibre-Web results

Table 6.10 summarizes the change detection results. The change detection algorithm for Calibre-Web has been executed using the action description. The states that have not been reached due to state model inference restrictions (action derivation and filters strategy, or depth limit) are not included in the table.

Table 6.10 Calibre-Web change detection qualitative results

Calibre-Web	Summary of detected changes
v0.6.18 to v0.6.19	<ul style="list-style-type: none"> - ALL states: The static left menu has changed with a new Downloaded Books item - Edit E-mail Server Settings form: has changed the default email mail.example.com to mail.example.org - Tasks table: has changed with a new Actions row - Edit User admin form: has changed the Kindle text to E-Reader - admin’s profile form: has changed the Kindle text to E-Reader - Add new user form: has changed the Kindle text to E-Reader

	<ul style="list-style-type: none"> - Edit Users table: has changed the Kindle text to E-Reader. This change is also detected when the table elements are selected - Users table and E-mail Server Settings panel: has changed the Kindle text to E-Reader. Moreover, the <i>E-mail Server Settings panel</i> has changed with less text information - Allowed Tags row element: In the Edit Users table, after clicking the Allowed Tags row element, the Change Detection algorithm detects a state as added and removed instead of changed. This occurs because this table element does not contain an action description, and actions are not detected as corresponding actions. - UI, Basic, and Database Configuration panels: are included as new states in the model because the E-mail Server Settings panel has changed with less text information. This is indirectly provoked because the resolution and the number of visible elements affect the state model inference process
v0.6.19 to v0.6.20	<ul style="list-style-type: none"> - Add New User form: has capitalized the page title, has changed the E-Reader text to eReader, and has changed the E-mail text to Email - admin's Profile form: has capitalized the page title, has changed the E-Reader text to eReader, has changed the E-mail text to Email, and has added admin@example.org as default email - Edit User admin form: has changed the E-Reader text to eReader, has changed the E-mail text to Email, and has added admin@example.org as default email - Edit Users table: has changed the E-Reader text to eReader, has changed the E-mail text to Email address, and has added admin@example.org as default email. This change is also detected when the table elements are selected. Moreover, the new email admin@example.org is now clickable and editable in a new SUT state - Users table and Email Server Settings panel: has changed the E-Reader text to eReader, has changed the E-mail text to Email, and has added admin@example.org as default email - Default Visibilities for New Users configuration: has changed by capitalizing the checkbox text options - Default Settings for New Users configuration: has changed by capitalizing the checkbox text options. This change is not visual in the state but exists at the DOM level

	<ul style="list-style-type: none"> - Basic Configuration panel: has changed with a new “Security settings” option. This change is detected in 4 subsequent states - Security settings option panel: has been added to the new SUT version - Edit Email Server Settings form: has changed the E-mail text to Email. Due to the change in the action description, the state is detected as added and removed instead of changed. - Allowed Tags row element: In the Edit Users table, after clicking the Allowed Tags row element, the Change Detection algorithm detects a state as added and removed instead of changed. This occurs because this table element does not contain an action description, and actions are not detected as corresponding actions.
v0.6.20 to v0.6.21	<ul style="list-style-type: none"> - Basic Configuration panel: has fixed a typo text from Security settings to Security settings. This change is detected in 4 subsequent states. This GUI typo issue was reported and fixed by the project contributors⁷ - Security settings option panel: is detected as added and removed due to the change in the action description - LogFile Configuration dropdown: adds the new default value /config/access.log for the logfile location - Feature Configuration dropdown: changes the checkbox text option Please ensure that users also have upload permissions - Edit User admin form: has changed by updating the dropdown Language to include the Português option. This change is not visual in the state but exists at the DOM level - admin’s Profile form: has changed by updating the dropdown Language to include the Português option. This change is not visual in the state but exists at the DOM level - Add New User form: has changed by updating the dropdown Language to include the Português option. This change is not visual in the state but exists at the DOM level - Default Settings for New Users configuration: has changed by updating the dropdown Default Language to include the Português option. This change is not visual in the state but exists at the DOM level

⁷<https://github.com/janeczku/calibre-web/issues/2811>

- **Edit Users table:** has changed by updating the select Language to include the Português option. This change is not visual in the state but exists at the DOM level. This change is also detected when the table elements are selected
- **Users table page:** contains a non-visible widget that contains a system version change. This change exists and has been detected at the DOM level
- **Allowed Tags row element:** In the Edit Users table, after clicking the Allowed Tags row element, the Change Detection algorithm detects a state as added and removed instead of changed. This occurs because this table element does not contain an action description, and actions are not detected as corresponding actions

Based on the evaluation of the results obtained with this complex Calibre web application, we can consider the challenging functionalities that negatively affect the state model inference and change detection process, and the further improvement ideas for the analysis of the detected changes. We observe that some results share similarities with the OBS-Studio application regardless of being different systems.

Calibre challenging results

In the delta version from v0.6.18 to v0.6.19, the static left menu of the web application has changed with a new Downloaded Books item. Similar to the OBS-Studio results, this change in the static menu does not interfere with the state model inference process itself but introduces change detection *noise* in all the states of the merged model. This change detection *noise* implies that all states will be marked as changed. Then, the user needs to analyze all states to determine if there are changes or not.

Also, in the delta version from v0.6.18 to v0.6.19, we found a challenge with the **UI, Basic, and Database Configuration panels**. Due to environment resolution and visibility of web application widgets, the inference process with v0.6.18 was not able to discover 10 states that were discovered in the incremental version v0.6.19.

Calibre further improvements ideas based on results

In the delta version from v0.6.20 to v0.6.21, the inclusion of the Português option as a language is not directly visible to the user when inspecting the merged graph. It has been necessary a complementary DOM analysis to validate the existence of this change. As mentioned for OBS-Studio, a descriptive message indicating the changed widget could help users recognize the changes easily.

The pixelmatch technique for highlighting GUI changes has had better results than for OBS. Nonetheless, in some states with tables, the pixel comparison results do not adequately assist the user in visualizing the specific GUI changes. Highlighting the widget's position on the screen with a rectangle can help the visual comparison.

In all delta versions, the **Allowed Tags row element** action has been tracked as added and removed state instead of changed because the action description remains empty. It is necessary to improve the state model inference process to use other web widget identifiers to be included in the description of actions.

MyExpenses results

Table 6.11 summarizes the change detection results. The change detection algorithm for MyExpenses has been executed using the state + action identifier. The states that have not been reached due to state model inference restrictions (action derivation and filters strategy, or depth limit) are not included in the table.

Table 6.11 MyExpenses change detection qualitative results

MyExpenses	Summary of detected changes
v3.6.6 to v3.6.7	<ul style="list-style-type: none"> - Settings Data panel: has changed by adding a new Unmapped Transactions option - Parties panel: has changed by removing the merge option that should only be shown if there are at least two parties - Select Category panel: has changed with two new Expense and Income filter options - New Main Category panel: has changed with two new Expense and Income filter options - Select Category-Help panel: has changed with a new Expense/Income FAQ option - Discard confirmation panel: has been detected as added when the user does not complete the creation of a new transaction template. Although there exist code changes that modify the back behavior and discard panel, we consider this detected change a false positive because we were not able to reproduce this functionality
v3.6.7 to v3.6.8	<ul style="list-style-type: none"> - Premium feature - Split transaction panel: has changed with a new radio button option - Premium feature - FinTS panel: has changed with a new radio button option

	<ul style="list-style-type: none"> - Premium feature - Budgeting panel: has changed with a new radio button option - Premium feature - Scan receipt panel: has changed with a new radio button option - Backup folder dialog: has changed because it shows a Java object string instead of the backup folder directory. We reported this bug in the project repository, which was resolved by the project developer⁸ - False positive - Typing Date: The dynamic date text detects a change due to an inadequate abstraction that does not match the additional after text
v3.6.8 to v3.6.9	<ul style="list-style-type: none"> - Categories panel: has changed the button Setup default categories to Transfer - Select Category panel: has changed the button Setup default categories to Transfer - Select Category unselected checkbox: now transits to a newly added state with the Transfer button - Backup folder dialog: continues showing another Java object string instead of the backup folder directory

Based on the evaluation of the results obtained with this complex MyExpenses Android application, we can consider the challenging functionalities that negatively affect the state model inference and change detection process, and the further improvement ideas for the analysis of the detected changes.

MyExpenses challenging results

In all versions, the hierarchical composition of some GUI elements was made up of multiple widgets. For instance, a button was composed of a clickable `LinearLayout`, a non-clickable `RelativeLayout` child, and a non-clickable `TextView` grandchild that contains the widget description. This causes some actions to have empty descriptions and cannot be used as action identifiers.

As observed in OBS and Calibre systems, using the action description for GUI change detection is a necessary approach when a delta increment affects a lot of states since it is not dependent on origin state changes. Although the state + action identifier is a valid approach for MyExpenses because the system versions evolve with many small delta increments, it will be beneficial to extract the description of the widget child during the model inference when a clickable Android element does not contain any description.

⁸<https://github.com/mtotschnig/MyExpenses/issues/1378>

MyExpenses further improvements ideas based on results

Similar to Calibre, the pixelmatch technique for highlighting GUI changes has had better results than OBS. Nonetheless, in some states with a lot of text, the pixel comparison results do not adequately assist the user in visualizing the specific GUI changes.

Answer to the research question

In order to answer Delta-RQ2: *Does the ChangeDetection tool detect delta GUI changes when using inferred state models?*, we conducted a manual qualitative evaluation of detected GUI delta changes with the OBS-Studio, Calibre-Web, and MyExpenses applications. Our results have demonstrated positive GUI change detection results and the potential to uncover GUI failures. Consequently, we reject H_0 : *The ChangeDetection tool does not detect delta GUI changes using inferred state models.*

Our proposed approach of using inferred state models addresses existing state-of-the-art limitations in GUI change detection to detect and highlight GUI change transitions for a diverse range of desktop, web, and mobile systems. Moreover, our tool provides an interactive interface with a merged model that users can easily use to visually analyze GUI changes. These ideas could help to pave the way to a delta GUI change detection standard that streamlines the efforts of developers, testers, and other project contributors in identifying functionalities that have been removed, added, or modified.

Nonetheless, as a result of novel research, our findings with complex GUI applications also expose challenges we will need to address in the future to improve the delta GUI change detection technique using inferred state models.

6.2.5 Threats to validity

This section mentions some threats that could affect the validity of our study [297, 232, 85].

- **Construct validity (Systematic mapping):** The decision on exclusion and inclusion criteria during the systematic mapping of the literature may be unintentionally biased by the researchers' knowledge and judgment. This is mainly because some techniques reported in studies that aim at repairing test scripts or detecting cross-browser and cross-device differences are strongly connected to the delta GUI change detection topic. Moreover, in other studies, it is not clear how changes are reported with textual or visual information. To mitigate this threat, we have documented and shared the process results, as well as reported other interesting related work studies.

- **Construct validity (Change Detection tool):** The state model inference was limited to 3 click-action lengths to control the number of discovered states and allow the inference of a partial but complete state model. However, this causes certain internal states of the application to not be inferred in the model, and the change detection approach will not be able to detect and highlight changes in these states.
- **Content validity (ChangeDetection tool):** The qualitative evaluation to determine the accuracy of the detected GUI changes was performed manually by inspecting the diverse SUT versions from the open-source software repositories. Although the results are described in a complete document, human error is possible in determining these results.
- **Internal validity (Systematic mapping):** The results of the systematic mapping of the literature search query may not retrieve all the relevant research works related to the GUI change detection topic. To mitigate this threat, we used Scopus, the largest database of peer-reviewed scientific literature, and we validated the employed terminology with a small set of relevant works found during our previous research [239].
- **Internal validity (ChangeDetection tool):** A unique researcher, with experience in the TESTAR state model inference, configured the abstraction strategy of the SUT objects. This introduces subjectivity and potential bias based on their tacit knowledge and decisions. Different researchers may consider that using a different abstraction strategy is more adequate.
- **External validity (ChangeDetection tool):** We use one open-source Desktop, Web, and Android application to conduct the study. We have demonstrated the potential of integrating the change detection approach to detect and highlight GUI changes between delta versions. Moreover, we mentioned diverse inference and abstraction challenges that affect this delta GUI change detection approach. Nevertheless, it is necessary to extend the research with diverse types of applications to be able to generalize the results.

As an additional external validity threat, all the state models used in this study are inferred using the TESTAR tool. For this reason, it is necessary to investigate other state models of event flow graph tools to extend and improve the ChangeDetection tool.

6.3 Summary

This chapter presents a delta GUI change detection approach that utilizes automated inferred state models to identify and highlight differences in the GUI between different SUT versions. Through a systematic mapping of the literature process, we have identified a gap in existing techniques or tools capable of automatically detecting and highlighting delta GUI changes for a diverse range of applications in desktop, web, and mobile systems. Furthermore, our examination indicates that outside web systems, delta GUI change detection techniques are mainly researched in regression testing studies that lack emphasis on reporting textual or visual GUI changes to end-users.

Recognizing the significance of integrating a delta GUI change detection technique into software repositories for software quality assurance, we provide a novel open-source tool to improve a GUI change detection approach that can detect and highlight GUI changes from 3 different software systems. The GUI change detection approach recursively compares two inferred state models from distinct software versions to mark the states and actions that have been changed, added, or removed. Subsequently, a merged graph technique allows the visualization of these changes in an interactive web interface. To substantiate our approach and contribute valuable insights into the software testing field, we have conducted an empirical evaluation using representative SUTs from real and widely adopted applications.

The empirical evaluation results demonstrate the feasibility of employing automatic inferred GUI state models for executing an algorithm that detects GUI changes, complemented by a merge graph technique that highlights these changes to end-users. These techniques are developed in a dedicated ChangeDetection tool, designed to operate independently with respect to the specific SUT technical requirements. However, it is important to note that, similar to any technique that relies on model inference, the efficacy of the ChangeDetection tool depends on the design of adequate inference and abstraction strategies.

Chapter 7

TESTAR: Scriptless testing in practice

As the world has increasingly adopted software systems as centric pillars of their every-day environment, the software industry, comprised of countless companies, faces constant pressure to shorten release cycles while maintaining high quality. Moreover, as customer demands and expectations grow, modern software becomes larger and more complex, often interconnected by numerous system components. The need for rapid development and testing cycles for these intricate software systems highlights the inadequacy of traditional testing methods, such as manual and scripted approaches.

Research plays a fundamental role in advancing knowledge and innovation, bridging the gap between theoretical understanding and practical application. In the realm of software testing, diverse research efforts focus on identifying the best strategies to prioritize test cases, developing reliable and robust mechanisms for maintaining test scripts, and investigating novel testing techniques. Some of these novel techniques, such as scriptless testing, aim to reduce efforts and complement existing industry testing approaches.

The TESTAR tool has demonstrated its effectiveness in identifying unexpected bugs and complementing companies' testing strategies through years of research and industry collaboration. The application of TESTAR in real-world software systems and the feedback from companies and stakeholders have been crucial in transforming it from a theoretical possibility to a practical, functional tool.

This chapter presents the academic experiences of collaborating with industry partners during this PhD thesis and the feedback obtained to improve the TESTAR tool for industry needs. Section 7.1 discusses the experience with Ponsse, a forestry vehicles and machinery company. Section 7.2 covers the collaboration with Prodevelop, a maritime port company. Section 7.3 details the work with the Kuveyt Türk bank. Section 7.4 presents the collaboration with the ING bank. Section 7.5 discusses the experience with Keen Software House and GoodAI gaming companies. Section 7.6 covers the experience with I.D.B. Telematica B.V.,

an enterprise solution company. Finally, Section 7.7 summarizes the industry-academia collaboration results.

7.1 Ponsse industrial experience in 2019

Ponsse Plc ¹ is one of the world's leading manufacturers of forest machines for the cut-to-length method, cutting the tree trunks in the forest to lengths that suit their intended use. At the same time, the information systems inform the end users of the types and quantities of timber they will next receive from the forest.

7.1.1 Ponsse Opti4G desktop SUT

The SUT of this study is the Ponsse Opti4G control system of the forestry machine. Opti4G has been in the market for a long time, and updates are still being delivered. Opti4G runs on a specific rugged PC hardware with a touch screen and Windows Embedded operating system. The long lifetime of the products results in a lot of legacy code, and the specific hardware with touch screen involves non-standard GUI specific code, introducing challenges for TESTAR and GUI test automation.

7.1.2 Ponsse study with TESTAR

Figure 7.1 illustrates the high-level architecture of the Ponsse environment. The components from inside the Ponsse network are located on Ponsse premises and connected to the Ponsse internal network services. The other TESTAR repository and email service components are outside the internal network of Ponsse.

TESTAR is developed on a public GitHub repository. The branch used for Ponsse-specific development is not public for security reasons. Most of the modifications have been SUT-specific instructions for TESTAR, but some generic features, such as handling multiple SUT processes and using image recognition to find specific widgets, have been merged into the master branch of TESTAR.

The test execution environment runs a Windows Embedded 32-bit operating system on rugged PC hardware with a touch screen. There is a CI agent running and connected to the CI server.

Ponsse is using an internal installation of Visual Studio Team Services for CI. There are two triggers for the CI server to start executing the TESTAR build definition: 1) CI polls for

¹<https://www.ponsse.com>

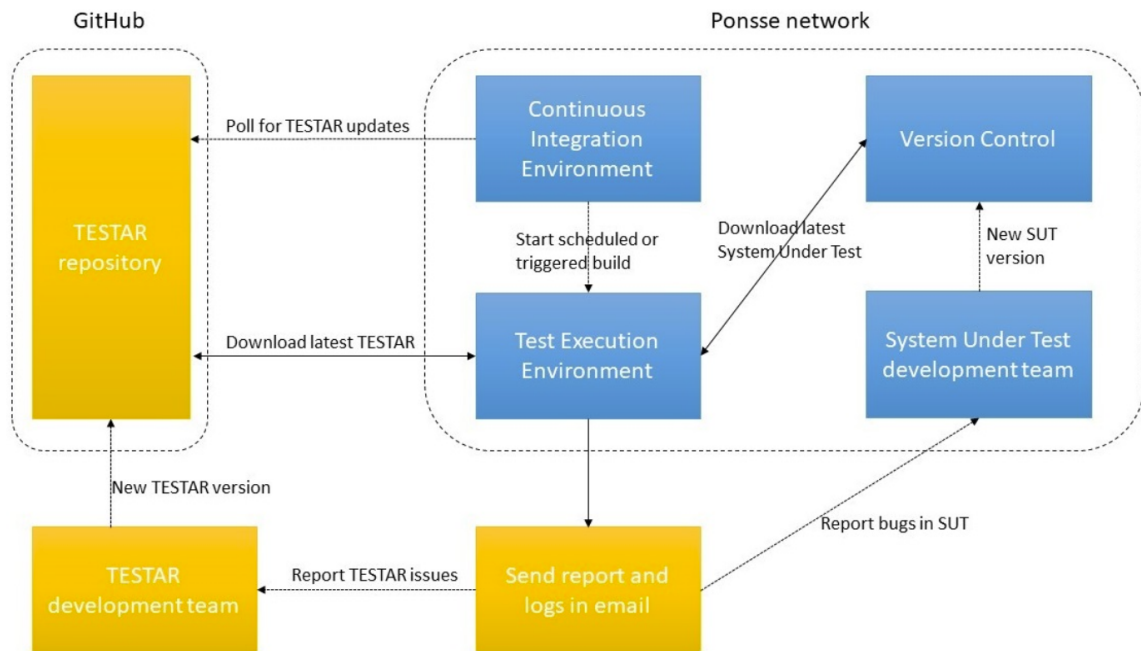


Fig. 7.1 The architecture of the continuous TESTAR piloting environment

TESTAR changes (commits) on GitHub, and 2) Scheduled trigger every office day at 6 am for testing the nightly SUT build.

The build definition for the TESTAR CI pipeline had the following steps that are executed by the CI agent:

1. Download and install the latest SUT version,
2. Download and build the latest TESTAR tool source code from Github (specific Ponsse branch),
3. Scripts for setting up a test user account for SUT,
4. Execute test scenarios with various TESTAR settings and given number and length of test sequences,
5. Archive and send the output folder of TESTAR as an email attachment to tool developers (test reports and logs).

The TESTAR tool developers analyze the test reports and logs to improve the tool and SUT-specific GUI exploration instructions. When suspicious SUT behavior is observed, it is reported to the SUT development team.

Supporting SUTs that invoke a child process

By default, the execution of a SUT was started up by TESTAR using the path that contains the executable file, invoking a Windows function that will return the *process* handle of the SUT process that allowed the tool to obtain the identifier of the SUT process, *pid*. Then, to obtain the GUI state information (i.e., the widget tree and all the widget properties) through the Windows Accessibility API plugin, TESTAR needs the *window handle*. To find the corresponding window handle, TESTAR checks all the existing window handles that are children of the Windows Desktop to find the one that has the same *pid* as the SUT process.

This Ponsse case study presented a challenge where the Ponsse Opti4G SUT started with a single process in a main window but later invoked a child process with additional window states and finished the previous main window. As a result, TESTAR was initially unable to recognize the widgets from the child process, only detecting those from the main window until it was closed. TESTAR was updated to handle SUTs that invoke child processes at runtime to address this. When a new process was detected during testing, the tool checked if the process had a GUI window in the desktop foreground that matched a specific process regex name. In such cases, TESTAR considered the child process *pid* the new main one and updated the *window handle* to obtain the GUI widgets from the new SUT process.

Supporting image recognition interaction

TESTAR uses the Windows API to obtain the screen coordinates of GUI widgets and the Java AWT Robot library to generate native system input events for click and type actions. During this Ponsse case study, a second challenge arose when executing GUI actions in the test environment. Although TESTAR could obtain the screen coordinates of the Ponsse Opti4G GUI widgets, the executed actions were misaligned due to resolution issues in the Windows Embedded 32-bit operating system. To address this, the SikuliX image recognition library was integrated into TESTAR, enabling the tool to capture the image of the desired widget and rely on SikuliX's capabilities to execute GUI interactions accurately. In subsequent versions of TESTAR, this resolution issue was also resolved in Windows systems by obtaining the display scale of the native monitor that renders the GUI of the SUT.

7.1.3 Results

The continuous approach enabled independent development of the tool being evaluated with the industrial SUT, reducing the collaboration effort required from the industrial partner. Triggering a new test run at Ponsse CI whenever a new commit of the TESTAR tool was pushed into the Ponsse-specific branch provided quick feedback for the TESTAR developers.

During this collaboration, TESTAR addressed two technical challenges in testing the Ponsse Opti4G control SUT. First, the tool implemented a multi-process solution to detect when the SUT invoked new child processes, allowing TESTAR to connect and extract the SUT's state. Second, a Sikulix image recognition solution was integrated with Windows API capabilities to execute actions within the test environment.

7.2 Prodevelop industrial experience in 2020

Prodevelop² is a Spanish company located in Valencia with an extensive network of clients in Europe, Africa, America, and Oceania. From the beginning, Prodevelop has specialized in Geographic Information Systems and its application to maritime transportation, especially in the port domain.

7.2.1 Posidonia web SUT and testing process

The SUT used in this study is Posidonia Management, a web-based port management application developed and maintained by Prodevelop. Posidonia Management is conceived and designed to fulfill the management needs of different Port Authorities. The increasing port traffic and high competitiveness of the international market lead to increasingly complex systems. It is in this context that Posidonia Management, as a complete management system, can improve the efficiency, productivity, and competitiveness of a Port Authority.

Until a few years ago, Prodevelop followed the waterfall development cycle, but in the last few years, Prodevelop has oriented its development practices towards a more agile development cycle, with more frequent product deliveries, weekly in some products.

Continuous integration [182] (CI) is a process that focuses on increasing the client value through developing, updating, building, and testing the software product as often as possible, for example, after each code commit or once a day.

The CI process of Prodevelop is made up of a series of linked and interrelated steps, illustrated in Figure 7.2. The process begins when the Quality Assurance (QA) team configures the automatization orchestrator server Jenkins³, a free and open source automation server that can be used to build, test, and deploy software, facilitating continuous integration.

In parallel, the Business Analyst will gather the project requirements and analyze them to obtain the specifications of the system. Based on this specification, on the one hand, the testers will use TestLink⁴ to define the Acceptance/Functional test, and on the other hand, the

²<https://www.prodevelop.es>

³<https://www.jenkins.io>

⁴<http://testlink.org/>

Developers will develop the system and create the Unit Tests. These tests will be evaluated by the task of Jenkins, which performs the build of the deliverables.

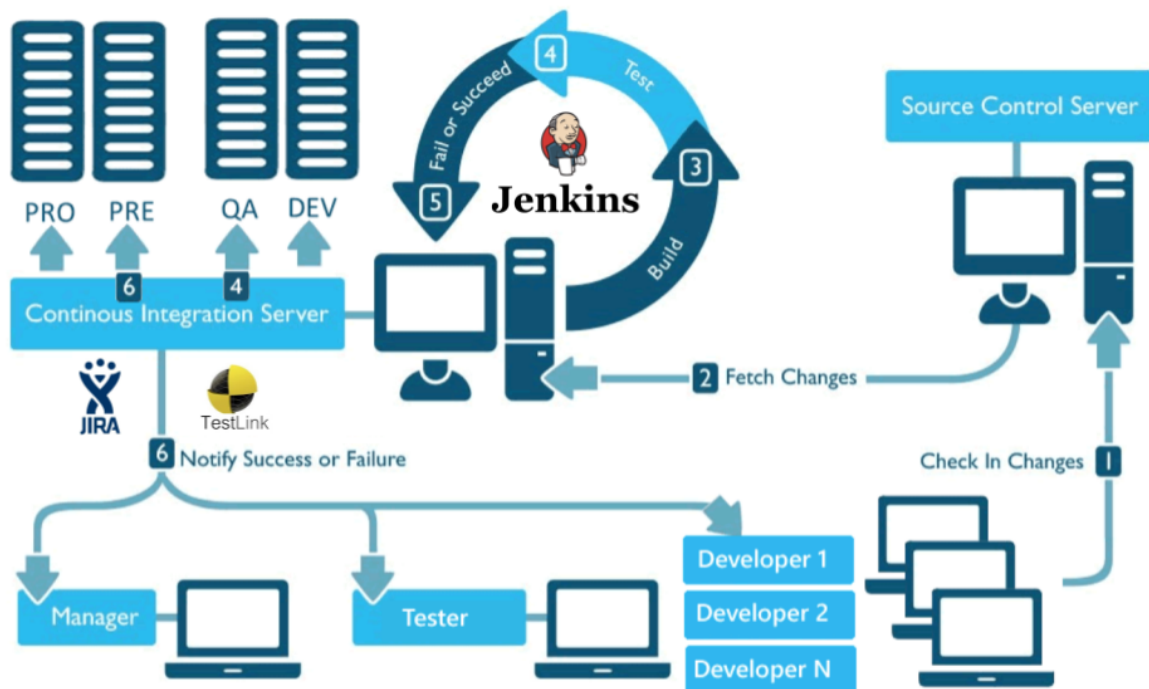


Fig. 7.2 Prodevelop CI/CD Pipeline

This automated process starts each time the developers commit source code to the repository. When a project that is assigned to a continuous integration environment receives an update of the source code, the Jenkins application will execute software code testing tasks: Static analysis, build, and unit testing, to validate and compile the new source code.

If the build tasks in Jenkins end with the result “OK”, the new version of the application will be deployed in the Quality Assurance (QA) environment, and the acceptance/functional tests are executed manually. If the tests are passed, the application will be deployed in the User Acceptance Test (PRE) and/or Production (PRO) environment, which are located in the Client’s own environments. The number of environments and deployment procedures are subject to the specific requirements of the Client.

In the case that any of the tasks that should be executed in Jenkins ends with "NOK" results, Jenkins informs the Developers detailing which test or tests have failed. In addition, Jenkins will generate an Incident-Ticket in Jira⁵, an issue tracking and project management software, with all the necessary information, including also the phase of the process and

⁵<https://www.atlassian.com/es/software/jira>

specifically the test that fails, so that the Project Manager follows-up until the incident is resolved.

To ensure the quality of the software, Prodevelop relies mainly on functional testing. The QA staff assigned to a project defines functional test cases for each requirement and scenario using the TestLink tool. These test cases are manually executed by the QA team when a new release is ready. A report with the results is generated and sent to the project manager to decide actions to be taken. On the other hand, developers are in charge of defining unit and static tests that are executed automatically.

The manual execution of functional tests is very time-consuming as they have to be executed for each new delivery. The automation of these tests is one of the short-term objectives of Prodevelop. Another important issue that needs to be improved is the time needed to solve an error. Since Posidonia is a large product with several million lines of code and with several developers involved throughout the life of this product, a lot of time is spent looking for the origin of the problem.

To facilitate error detection and root cause analysis, the Posidonia Management application is instrumented with the intention to detect and debug all behavior that is identified as an exception. All these exceptions are written in a log file with information about the method where they occurred (the specific class and package they belong to) and details about the exception that has been detected. This internal error information is added incrementally in the background log using local timestamps.

7.2.2 Prodevelop study with TESTAR

Prodevelop is trying to achieve a high level of software quality by innovating its development processes. As indicated, the functional tests that are executed manually involve a high cost of running the tests. For this reason, only a subset of them is executed in each release. So the objective of the study is clear: *integrate TESTAR into the current CI pipeline to automatically test Posidonia when the life cycle requires it and evaluate the performance.*

With TESTAR integrated into the CI pipeline, every time a new version is released and a nightly build is made, the following steps are taken:

- First, new test sequences will be *generated* with TESTAR to explore and verify the robustness of the application using the desired oracles and protocols. Depending on the configuration used, TESTAR can be steered to explore specific parts of Posidonia.
- Second, if a failure is detected, Prodevelop must verify that it is not a false positive, inspecting the sequence that found the failure. If it is not, all the logs generated during the test run should be filtered by the timestamps of the failure finding sequence, saved

in a database, and documented in TestLink. Then, a Jira ticket will be created with linked information about these results to be reviewed in the future.

To start the integration, Posidonia was tested with the default set-up of TESTAR to generate: test sequences, TESTAR logs, HTML test reports, and GUI screenshots. All these artifacts generated by TESTAR were analyzed by Prodevelop. It was found that before the integration into CI could be realized, the following TESTAR extensions and improvements had to be implemented first:

1. Enable invocation of TESTAR through the CLI (Command Line Interface). This means that the configuration dialog should be disabled, and instead of passing the test settings in a local file, they should be passed as parameters of the CLI command.
2. Enable TESTAR to correctly detect SUTs that have multiple processes handling the GUI or that the GUI process changes at run-time. Posidonia runs in a browser that starts with two main processes to which we should connect to properly verify the defined oracles.
3. Enable distributed execution of TESTAR by providing a remote API. This feature is fundamental if we want to integrate TESTAR into the CI methodology or any other distributed process for that matter.
4. Enable the synchronization of the logs produced by Posidonia with those of TESTAR. In order to find the root cause of the errors, it is important to be able to analyze the logs generated by Posidonia together with TESTAR logs. This information is needed by Prodevelop developers to understand and replicate the error.

Supporting the execution and configuration of TESTAR through CLI

To allow the TESTAR tool to be integrated into a CI pipeline, a new configuration option was added in addition to local settings files. When starting TESTAR through a CLI, the configuration can be passed on as parameters. This way, any configuration setting can be overwritten through CLI, also disabling the GUI. This feature makes it easier to put TESTAR configuration into the settings of the CI job that starts TESTAR execution and change it from the CI tool.

Supporting SUTs with multiple GUI processes

By default, TESTAR started the execution of a SUT using the path containing the executable file, invoking a Windows function that returned the *process* handle of the SUT, allowing

the tool to obtain the process *pid*. During the Ponsse case study, TESTAR was improved to detect and connect with an invoked but single child process. While this implementation worked well for the Ponsse Opti4G SUT, it was insufficient for the Posidonia SUT.

Posidonia did not run in a single process. Instead, it started the execution with two GUI related processes. Some elements of one of these processes used warning pop-ups or lists of items. This prevented TESTAR from recognizing all the widgets of the two existing processes. Therefore, TESTAR had to be enhanced to deal with SUTs that start with multiple GUI processes at run-time. When a SUT starts up multiple processes, the tool does not have one main *pid* but has a list of *pids* (i.e., including the child *pids* of this main *pid*). In such cases, TESTAR iterates over all the elements in the list to be able to get the GUI properties and information for each *pid* and merge them into one widget tree.

Supporting TESTAR execution with a remote API

In order to integrate TESTAR into the Posidonia CI test cycle, the next step was to design a CI architecture [204] in which TESTAR can be invoked remotely in a distributed manner. First, suitable technologies were required for the communication between (1) the CI server that launches the test execution, (2) the server that contains TESTAR, and (3) the server that executes the SUT.

Thinking about future deployments and enabling TESTAR execution in a test server environment, a Spring boot application was developed with an Apache Tomcat servlet that provides an API for TESTAR settings. Prodevelop offered the initial version of the API that was updated by the TESTAR developers with other necessary requirements, such as new settings parameters for remote login (instead of coding the user login inside the TESTAR Java protocol) and additional configuration options for the initialization of the GUI state model that is built during testing.

With the default implementation, the web API instance should be running in the same directory as the TESTAR tool. Subsequently, when receiving a POST request that is compatible with the TESTAR settings from the CI orchestrator, the contents of the web parameters will be parsed into CLI instructions. The flow of the invocation from the CI pipeline is depicted in Figure 7.3. The main steps of the functionality are:

1. Upon receiving a web POST request, the Posidonia CI orchestrator will send the desired configuration settings to run TESTAR. Only a couple of parameters were needed in the request payload.

2. The remote API is running in the same directory with TESTAR binaries to receive the requests and transform the parameters into a TESTAR configuration that is executed through the CLI.
3. If all the parameters are correct, TESTAR execution will start, and a response will be sent back with the output information printed by TESTAR on the CLI, which includes the test results, the path of the generated sequence, and a timestamp to indicate when the sequence began.
4. If more detailed information about any sequence is required, a request will be sent indicating which sequence we want to obtain the resources from.
5. Then, a response with the desired resources will be sent back.

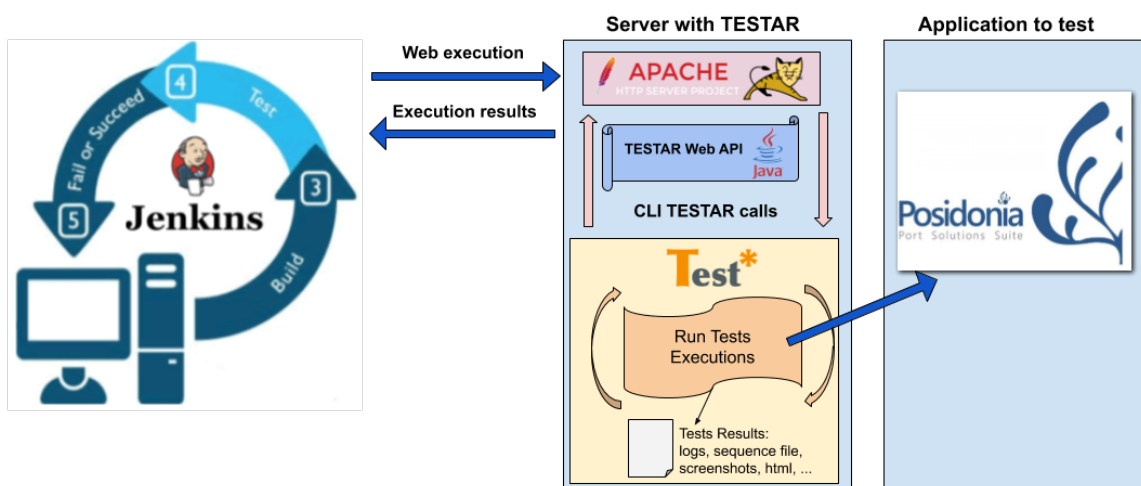


Fig. 7.3 Integration of TESTAR through an API in a distributed environment

Improving the output results and the structure of TESTAR logs

The various logs and resources created by TESTAR could offer a large amount of information about the different GUI elements detected by TESTAR in the different states that conform the SUT. However, these files were not stored in a suitable structure for the case study. All the resources were stored in their corresponding directory (logs, sequences, HTML reports, screenshots), but they were stored incrementally according to the sequence number without taking into account the execution of TESTAR. With this structure, the objective of synchronizing Prodevelop and TESTAR logs could not be achieved, and therefore, it had to be changed.

The solution was to create an *index log* and restructure the output directories according to the timestamp in which TESTAR was launched, in addition to the sequence number. Posidonia can then use this index every time it needs to obtain GUI information from TESTAR logs. Using its own logs and its own timestamps, Posidonia will filter the desired sequence in the TESTAR index and will be able to obtain the resource path with all the required information.

In Figure 7.4, we can see that Posidonia creates its own logs based on its internal state. If an error occurs, a timestamp will be used to find the matching event from the TESTAR index log to obtain all existing resources and verify which front-end GUI action produced the back-end error.

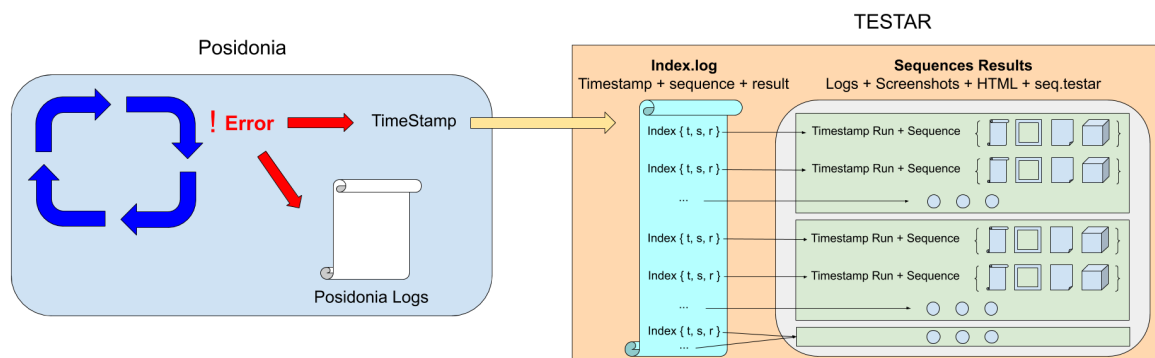


Fig. 7.4 Posidonia and TESTAR Logs Structure

7.2.3 Results

Due to various circumstances, the SUT Posidonia evolved into a "maintenance only" phase, and Prodevelop decided not to make any changes to their existing testing processes because there are hardly any changes to the SUT anymore. This meant that, unfortunately, we could not really evaluate the performance of TESTAR in a real CI environment.

To try and get some data, we simulated a test with Posidonia by running TESTAR during 4 nightly builds for 12 hours with random action selection protocol and a configuration of 30 sequences of 200 actions each night. Unfortunately, the SUT did not change in between, so the outcome of the runs could only differ due to the randomness of TESTAR. The runs showed that the CLI adaptations, the detection of multiple processes, and the distributed execution did not fail during long, unattended runs.

The outcomes of the test runs were:

- a total of 24000 actions in 120 test sequences

- 15 sequences resulting in *suspicious titles* (all found during the first run)
- 6 sequences resulting in *unexpected close* (all found during the first run)
- 0 sequences resulting in *unresponsiveness*

Analyzing these faulty sequences using the HTML report revealed that:

- 12 of the found *suspicious titles*-failures all lead back to a database connection error in Posidonia when TESTAR executed actions related to querying a port registry.
- the other 3 *suspicious titles*-failures lead to another database connection error in Posidonia trying to generate and obtain the expedient of port activity.
- the 6 *unexpected close*-failures were all false positives related to the fact that TESTAR tries to bring the SUT to the foreground.

The two errors that were found executed different database requests, and both were related to an error in the Posidonia database connection. Prodevelop was aware of these glitches in the software but decided not to fix them.

To validate the log synchronization, TESTAR and Posidonia logs were compared to check that the failure sequences found by TESTAR could be mapped to the internal error logs from Posidonia. The mapping was found correctly, and the names of the methods and classes that provoked the exception in Posidonia were meaningful with respect to the properties of the web elements on which the actions were executed. However, there was a delay of 5 - 10 seconds between timestamps. This is attributed to the time needed for the internal process to represent and detect the data at the GUI level.

The mapping did not only help to verify the synchronization of errors after the execution of a sequence, but also motivated us to investigate the possibility of synchronizing TESTAR with other possible internal logs in order to find a way to improve the action selection based on the available internal methods.

The results of this study are threefold. First, TESTAR has been extended with four new valuable features that will also be useful for other test environments (i.e., CLI invocation, multiple processes, remote testing, and log synchronization). Second, it was shown to be a useful complement to the existing testing practices and find failures. Third, some lessons, which are discussed in Chapter 9, were learned during this industry-academic collaboration.

7.3 Kuveyt Türk industrial experience in 2021

Kuveyt Türk Participation Bank Inc.⁶ is a private financial institution in Turkey that has been operating since 1989. It has 414 branch locations across Turkey and delivers a wide array of financial products and services to customers.

7.3.1 Kuveyt Türk Bank SUT and testing process

Kuveyt Türk Bank uses Selenium and Appium for regression testing of mobile and internet banking. These tools require manually defined test scripts to run relevant test scenarios in the development environment. The maintenance of scripts must be manually performed for each regression set, provoking high maintenance costs for the bank.

7.3.2 Kuveyt Türk study with TESTAR

The aim of this case study was to evaluate the use of scriptless testing and the TESTAR tool in order to reduce maintenance costs while increasing test coverage and testing the robustness of systems.

Originally, TESTAR was developed for testing desktop applications by using the accessibility API of the Operating System (OS) for extracting the GUI information [32, 277]. By considering the web browser as a desktop application, it could be used for testing Web applications too [36, 13, 172, 55]. During the industrial study described in this paper, TESTAR has been extended with support for using Selenium WebDriver instead of the OS accessibility API to obtain the state of the GUI.

Before applying TESTAR to the Internet banking application, we evaluated the tool on the public website of Kuveyt Türk Bank. The website is more static than the Internet banking applications and acted as an initial SUT to evaluate the logic and capabilities of the TESTAR tool. The website is available in multiple languages, but TESTAR was configured to stay in the English section.

When we started with the evaluation, TESTAR was used with the default Windows accessibility API to extract GUI information, including widgets, their locations, and other properties. It was quickly noticed that the accessibility API did not work well with modern Web applications. Without SUT-specific instructions, TESTAR did not recognize all the widgets, recognized available actions on widgets that were not visible or clickable, and continued testing after clicking links to external websites. Especially web pages that use DIV-elements or JavaScript to handle user interactions through event listeners were difficult

⁶<https://www.kuveytturk.com.tr/en/>

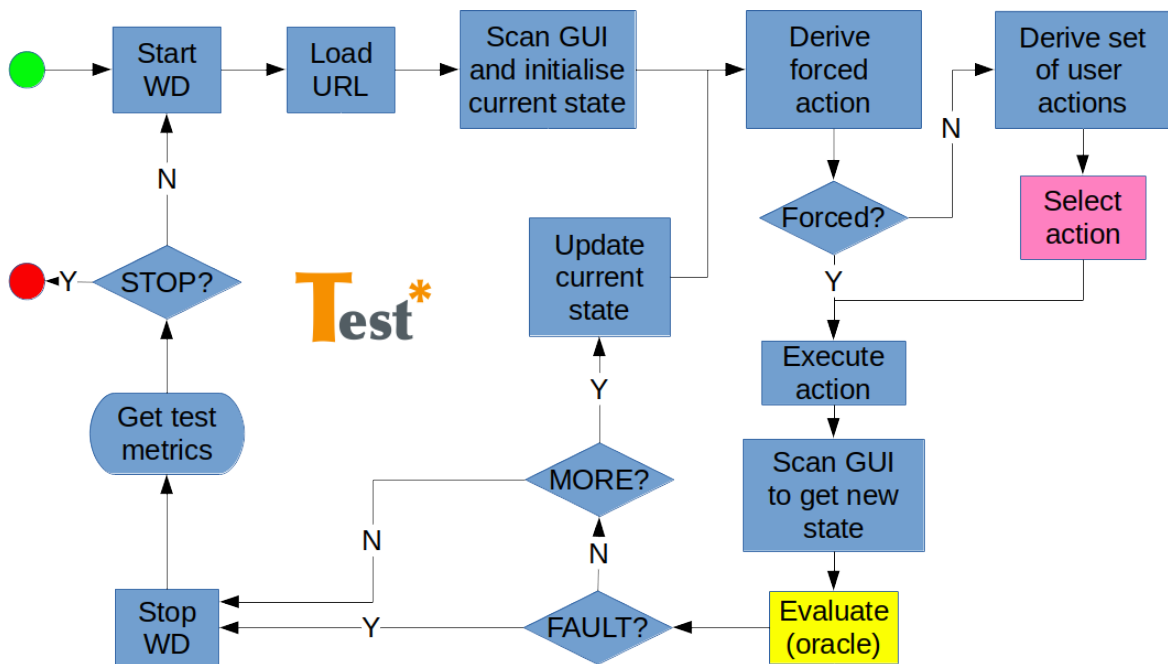


Fig. 7.5 TESTAR test cycle extended with forced actions

for TESTAR with the standard accessibility API: browsers do not disclose the existence (or absence) of these event listeners. Some of these issues could be solved with SUT-specific instructions in the TESTAR protocol class to carry out specific actions, but then these instructions were very specific to the website and would require maintenance if the site was changed.

To address these challenges, TESTAR was extended to include various functionalities for testing web pages. The accessibility API was replaced by support for Selenium WebDriver to get information about Web GUIs. To handle widgets that use JavaScript event handlers for user interactions, the Javascript functions to add and remove event handlers were extended dynamically with bookkeeping functionality using a Monkey Patching solution.

To restrict TESTAR from testing external Web pages, a SUT-specific protocol class was extended with:

1. Allow-listed domain URLs that are to be considered part of the SUT and
2. Block-listed extensions of resources, such as PDFs, that should not be tested.

Out-of-domain actions and URLs were prohibited, so that TESTAR did not select external links. If it ended up in a domain outside the SUT, a forced action, e.g., browser "go back", was triggered to return back to the SUT. Figure 7.5 shows the resulting operation flow of TESTAR.

7.3.3 Results

When evaluating TESTAR with WebDriver, significant improvements were achieved compared to Windows accessibility API. On the public website of Kuveyt Türk Bank, TESTAR with accessibility API found 35 correct widgets that could be interacted with, 155 false positives (widgets that actually could not be interacted with), and missed 14 false negatives (widgets that were not found but could be interacted with). TESTAR, with WebDriver and other web extensions, found 54 correct widgets, 5 false positives, and 2 false negatives.

The case study continued with testing the more complex Internet banking application in the internal development environment of Kuveyt Türk Bank. The initial configuration of TESTAR included passing the login and PIN code process and closing some pop-up warning windows. One detected challenge relevant to scripted test automation, as well, was a unique way to detect specific GUI elements, as element identifiers were dynamically generated and changing, and some other attributes were changing based on the selected language of the web page. Regardless of the challenges, TESTAR was able to find 2 relevant errors that would have been visible to the end users. The errors were fixed by the development team. Scriptless GUI testing complements the existing scripted approaches by covering also the less probable paths through the GUI. However, defining test oracles is more difficult for a scriptless approach.

In summary, we discovered that Windows accessibility API did not work well with web applications. TESTAR was extended to use Selenium WebDriver and other Web testing functionalities and successfully applied it to the industrial case study of the Kuveyt Türk Bank web services, finding 2 relevant errors from the internal development version.

7.4 ING industrial experience in 2022

ING⁷ is a European universal bank with more than 60,000 employees who serve more than 38 million customers in over 40 countries. In most of their retail markets, ING products include savings, payments, investments, loans, and mortgages. For their Wholesale Banking clients, ING provides specialized lending, tailored corporate finance, debt and equity market solutions, sustainable finance solutions, payments & cash management, and trade and treasury services.

⁷<https://www.ing.com/>

7.4.1 ING mobile SUT and testing process

ING predominantly emphasizes offering mobile banking services, highlighting a mobile app serving 5 million active end-users. To ensure this mobile SUT application functionality and reliability achieves high-quality standards, ING employs a combination of automated and manual testing approaches. The automated testing is handled by Espresso GUI test scripts developed and maintained by the Android development team. Although confidentiality does not allow further details on the Espresso test scripts, it is possible to reveal that these are the real set of test cases that the ING uses to test the SUT.

One of the main goals of the ING company is to reduce the costs of testing. The combination of manual GUI testing with the creation and maintenance of scripts to cover the whole application is too expensive. For this reason, the ING company was interested in evaluating the scriptless testing approach on their own commercial software instead of using open-source apps.

7.4.2 ING scriptless testing study for mobile applications

Although scriptless GUI testing of mobile applications is not well established in the industry, there are a number of academic (Android) scriptless GUI testing tools. ING decided to study existing tools that are open source and published in the last 5 years so that the supported Android version is not too old. The tools were evaluated by the following requirements:

- **Actively maintained tools:** Check whether the tool is still actively maintained.
- **Action Selection Mechanisms (ASM):** The action selection mechanism (ASM), or exploration algorithm, is a core aspect of scriptless testing as it determines how the actions are selected to generate test sequences. With an ineffective ASM, a large part of the application might stay uncovered and, therefore, untested, compared to a smarter or more suitable mechanism.

Existing ASMs vary in complexity and approach. The simplest, *Random (RND)* [155, 41, 2], randomly selects an action out of all possible actions in the current state. *Least executed actions (LEA)* [41, 2], also known as a frequency-based algorithm [3], prioritizes actions on UI elements that have been the least interacted with.

More advanced ASMs require some sort of memory or model to remember what has been explored and which actions have been taken. A state model can be used for this, but then a concept of state is required. *Random biased by predicted probabilities (RBPP)* [41, 156] uses a learned or mined interaction model to predict the probability of a user action and incorporate these probabilities into the random selection process.

Unvisited action first (UAF) [155], uses a state model to obtain a sequence of actions that leads to a state with unvisited actions. *Breadth-First-Search (BFS)* [103, 52] systematically explores the SUT by executing all possible actions in the current state before moving to the subsequent states. *Depth-First-Search (DFS)* [155], in contrast, focuses on deeply exploring one action path at a time by immediately following each new action as it is discovered. *Reinforcement Learning (RL)* [103, 52, 156] uses rewards to guide an agent in the process of learning what action to take under different circumstances. *Metaheuristic search-based optimization (MSBO)* mimics some natural process [165, 265] that improves over time in an attempt to search for an optimal solution. *Combinatorial (Com)* [3] algorithms try to generate sequences to maximize the coverage of n-way event combinations and minimize redundant execution of events.

- **Flexibility for adding a domain or SUT-specific information:** To get good coverage during automated GUI exploration, the scriptless testing tool might need a domain or application-specific information, for example, username and password for a login screen.

The flexibility to add domain-specific AUT information to the scriptless testing tool is an important aspect for the industrial take-up of a tool. However, because scriptless testing aims to reduce maintenance effort compared to test scripts, adding AUT-specific information to the tool and maintaining it should be as easy as possible for the tester and not create an additional maintenance burden.

Since scriptless GUI testing tools generate the test sequences during run-time, the tools require a way to know when and where to use the provided application-specific information. One option is to use triggered behavior, specifying a unique GUI element to recognize when and element locators to specify where to use the pre-specified data.

- **Concept of State:** Since the more advanced ASMs often use some kind of abstract state model for navigation, we also researched state abstraction for mobile AUTs. Unfortunately, many approaches, e.g., Sapienz [165], CrawlDroid [52], and DroidBot [155], do not explain in detail how they define and abstract the state of the SUT. The following state abstraction approaches have been used:
 - Available actions to define the state of the SUT [103].
 - Android Activity as the abstract state [23].
 - The same widgets on the screen (identical hierarchy tree of widgets) [155].
 - Identical hierarchy tree of widgets and identical widget attributes [113, 265, 276].

- Identical screenshots [41, 79].

Using Android Activity as the abstract state does not work well for dynamic Android apps as a lot of changes can occur within one Android activity. Identical screenshots would lead to state space explosion as any change in the pixels of the screenshot results in a new abstract state. A tester-defined number of widget attributes to be identical seems to be the best approach for abstract state definition in the mobile domain. For the tool evaluation, we checked whether any concept of state is supported.

- **Support for test oracles for failure detection:** Test oracles significantly contribute to test effectiveness and reduction of costs [174]. In scriptless testing, test oracles are more difficult to define, because the test sequences are generated on-the-fly, during the execution. In [28], test oracles have been classified into four categories.

Specified oracles use a specification of the SUT, defining what behavior is acceptable and what behavior is faulty. Specified oracles are not commonly used in scriptless testing, except when based on invariants or very specific system properties. In these cases, specified oracles can be leveraged in scriptless GUI testing and require only a little domain knowledge to be inserted as a test oracle.

Derived oracles are based on the information derived from artifacts, e.g., documentation, system executions, or other versions of the SUT. An example in GUI testing would be using inferred state models derived during scriptless testing to check the consistency of consequent versions of the same SUT [59, 8].

Implicit oracles rely on general or implicit knowledge to determine whether an application is in a faulty state or not. They can be applied to almost any SUT and require little maintenance if the SUT changes over time [28]. A common implicit online GUI test oracle is detecting when the SUT crashes, becomes unresponsive, or whether there are unhandled exceptions or other warnings or error messages in the GUI, system output, or logs during testing.

Human oracles are not automated oracles but rather support systems to make it easier for humans to determine the correctness of the SUT. Human oracles are not suitable for scriptless online oracles, as they would require a human to be present and interact with the scriptless testing tool during the test execution.

Table 7.1 summarises the studied tools.

Sapienz [165] is a scriptless Android testing tool designed for search-based (SB) objectives, aiming to generate diverse sequences with minimal length while maximizing code

Table 7.1 Summary of existing tools for scriptless mobile testing

	Sapienz [165]	DroidBot [155]	Humanoid [156]	Stoat [265]	DroidMate- 2 [41]	CrawlDroid [52]	AutoDroid [3]	AimDroid [103]
Year of publication	2016	2017	2019	2017	2018	2018	2018	2017
Actively maintained	Not OS	Yes	No	No	No	No	No	No
ASM	<i>MSBO</i>	<i>DFS,</i> <i>RND,</i> <i>UAF</i>	<i>RL, RBPP</i>	<i>MSBO</i>	<i>RND, LEA,</i> <i>RBPP</i>	<i>BFS, RL</i>	<i>RND, LEA,</i> <i>Com</i>	<i>BFS, RL</i>
Flexibility domain knowledge	Yes	Yes	No	No	Yes	No	No	No
Concept of state	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes
Failure detection	implicit (crashes)	no oracle	no oracle	implicit (crashes)	implicit (crashes)	implicit (crashes)	no oracle	implicit (crashes)

coverage and fault detection. Sapienz enables the integration of application-specific knowledge of the SUT and includes oracles to check for application crashes. However, since 2017, it has been further developed and used internally at Facebook.

DroidBot [155], and its extension Humanoid [156], is a model-based scriptless Android testing tool that builds a runtime model of the SUT, featuring several exploration methods like *Depth-First-Search*, *Random*, *Unvisited action first*, *Reinforcement Learning*, and *Random biased by predicted probabilities* (Humanoid only). Users can create or modify exploration strategies and inject application-specific information. Although DroidBot is actively maintained, it has limitations in integrating domain-specific information and lacks an oracle component. Humanoid, while open source, has not been maintained since 2019.

Stoat is a stochastic model-based scriptless Android testing tool [265]. It combines dynamic and static code analysis to create a stochastic finite state machine model of the SUT's GUI, with edges associated with probabilities for test generation. Stoat defines the SUT state as a widget hierarchy tree, where non-leaf nodes represent layout widgets and leaf nodes executable widgets. Nevertheless, it lacks flexibility for injecting application-specific information without significant source code modifications. Although Stoat was released as an open-source tool, it has not been maintained since early 2020.

DroidMate-2 is a scriptless Android testing tool [41] that allows testers to develop custom action selection strategies, with default options like *Random*, *Least executed actions*, and *Random biased by predicted probabilities*. It generates state and widget identifiers, abstracting non-SUT UI elements, and monitors GUI exploration performance by intercepting all API calls between the Android OS and SUT, requiring instrumentation. While it is a flexible tool that enables injecting application-specific information, DroidMate-2 only supports crash test oracles and has not been actively developed since 2019.

CrawlDroid [52] groups the widgets in a GUI state that it considers to be equivalent and uses a feedback-based exploration strategy that intends to trigger actions on groups that

tend to improve code coverage. *Breath-First-Search* has been implemented for comparison. However, the open-source project has not been maintained.

AutoDroid [3] is a tool for the automatic exploration of GUI-based Android applications. It uses a *Combinatorial* algorithm to maximize the coverage of event combinations. AutoDroid defines state based on the name of the currently running activity and the set of available actions. Available actions are determined based on the widget tree. Their latest tool, DeepGUIT, expands AutoDroid into a Deep Q-Network-based Android application GUI testing tool [63]. However, the paper was published after the evaluation, and at the time of writing, the tool was not yet available.

AimDroid [103] is a scriptless GUI testing tool for Android, using a mix of *Breath-First-Search* and *Reinforcement Learning* algorithm in the ASM. It uses available actions to represent the current state of the GUI and a two-level state model for ASM. It supports crash oracle, and a crash is uniquely identified by the error message and the crashing activity. However, it does not seem to support domain-specific knowledge.

There are also other tools, but these are not open source, for example, ComboDroid [287], FARLEAD-Android [146], and CrashScope [183].

Because none of the existing scriptless mobile testing tools matched the requirements, ING decided to collaborate in extending TESTAR [276], an open-source tool for scriptless GUI testing that supported web and desktop applications but not yet mobile applications. TESTAR was already integrated with WebDriver, and Appium uses almost the same API, so the integration was evaluated easier than extending the other tools that support only Android. In addition, TESTAR was already being evaluated at ING for testing web applications⁸, so there were additional synergies.

TESTAR scriptless testing tool solution for Android and iOS systems

The TESTAR tool was extended to support Android and iOS applications. Ideally, the tool should have a shared core for scriptless testing of both Android and iOS to minimize maintenance costs. Therefore, Appium⁹ was chosen to be integrated into TESTAR and to be used as the API to retrieve the widget tree and execute actions on the SUT. Appium is a test automation driver that implements the WebDriver API for mobile apps.

The GUI state that Appium returns is an XML document with all the Android or iOS widgets and their attributes in a tree format. This tree is parsed, and each element is saved as a widget together with its corresponding attributes. Although Appium claims that it abstracts away the OS from which the information is obtained, in practice, there are

⁸<https://medium.com/ing-blog/scriptless-gui-test-automation-at-ing-54c003649aa6>

⁹<https://appium.io/>

significant differences between iOS and Android. For example, Appium obtains different widget attributes for Android and iOS applications. Due to this difference, two engines were constructed, generating TESTAR state for Android and iOS separately.

Deriving available actions from the widget tree had to be implemented separately for both Android and iOS, as it depends on the widgets and their attributes.

- For Android, the actions *click*, *long-click*, *scroll*, *type*, *back*, and *system* actions have been implemented.
- For iOS *click*, *scroll*, and *type* actions have been implemented. iOS does not support *Long-clicks* and *back* actions. Also, Appium does not support *system* actions for IOS.

To determine which of the implemented actions are possible on which widgets of the state, each widget in the state is iterated over, and its attributes are inspected. For example, Android widgets carry the property *clickable*; if this boolean is true, we add the widget with the click action to the derived actions list. Unfortunately, not all actions have clear mappings between widget attributes and actions. In these cases, the widget class property can be used. For example, iOS class attribute *XCUIElementTypeButton* means the widget is clickable. Which classes should be mapped to each action may depend on the SUT and is therefore added into the TESTAR configuration.

Figure 7.6 illustrates the widgets detection and actions derived integration with the ING mobile application. Part 1 of the Figure (indicated with a red number 1) shows the screenshot of the SUT, and the augmented green dots indicate where TESTAR recognized a click action. Part 2 of the Figure shows a selectable list of the widgets of the widget tree. Part 3 shows the attributes and attribute values of the selected widget.

It is important to mention the difference in performance for Appium between Android and iOS. Executing operations on the SUT does not delay TESTAR in either of the operating systems. However, retrieving the state takes significantly longer for iOS because Appium retrieves all widgets of the GUI state in iOS, not only the visible ones. This leads to a massive performance hit. For Android, retrieving the GUI state takes, on average, 10ms on the application used in the validation. For iOS, it takes around 2000ms to retrieve the GUI state on the iOS version of the same SUT. This significantly slows down TESTAR execution for iOS applications because obtaining the GUI state is often used within the TESTAR test generation flow. Although the iOS version of the ING application was also tested, the poor performance of Appium on iOS made it infeasible to get statistically significant results for the experiments on iOS. Therefore, the following experiments were performed only on the Android version of the application.

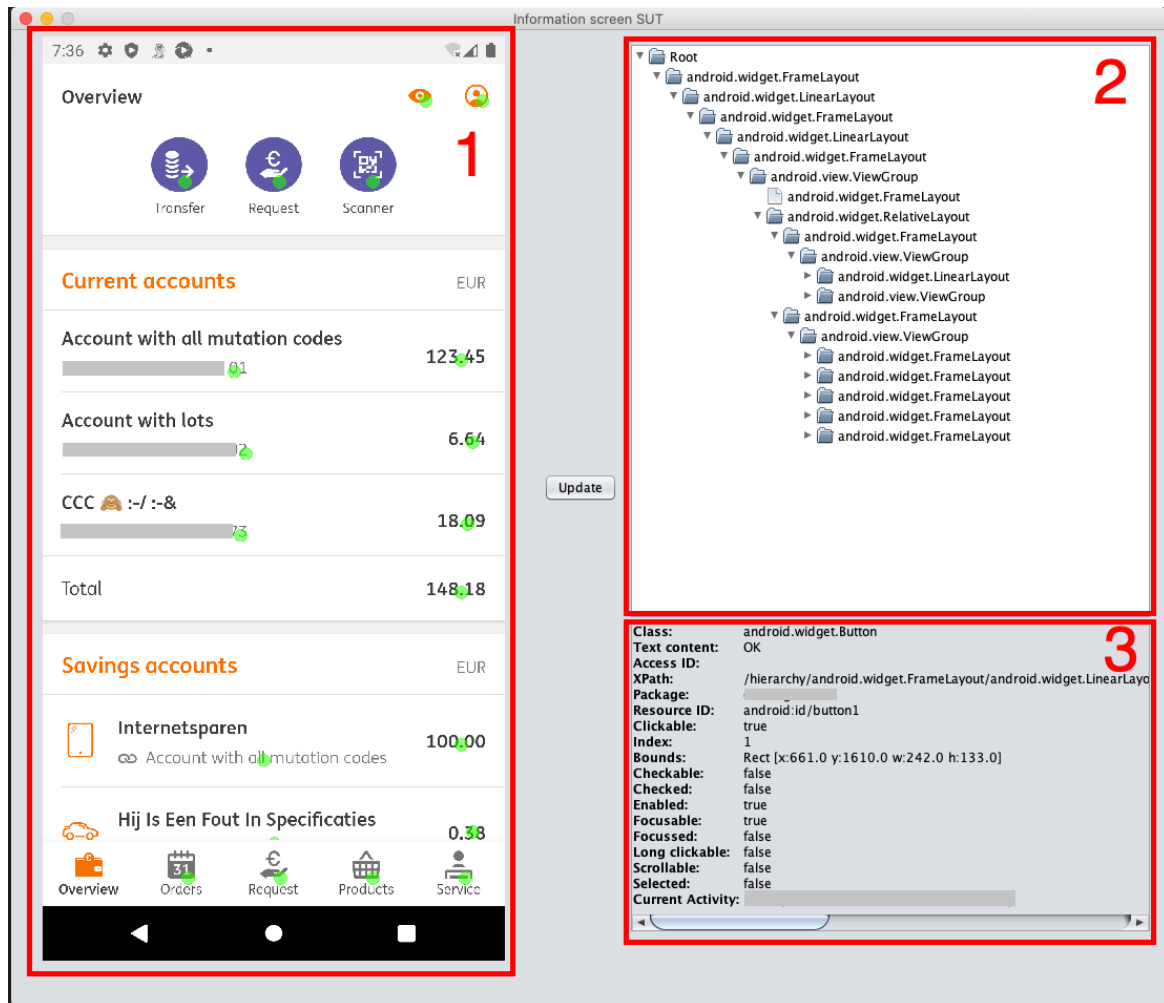


Fig. 7.6 TESTAR integration with the ING mobile application

7.4.3 Empirical evaluation

The empirical evaluation study was performed at ING, a large enterprise having a significant IT department due to the importance of digitalization. The mobile application of the ING that is used as the SUT in this case study has 5 million active end users. Therefore, it fits well for validating the performance of the mobile extension of TESTAR in an industrial setting. However, due to the type of the application and the company, some details are confidential, including the details about the bugs found.

In order to guide this scriptless testing for Android study, we have formulated the following research question:

- ING-RQ1: How do academic tools TESTAR, Stoat, and DroidBot contribute to the effectiveness of testing when used in a real industrial environment and compared to the current automated testing practices at the ING?

Cases of the study

To answer ING-RQ1, this study compares the performance of scriptless testing tools TESTAR, Stoat, and DroidBot and the existing Espresso GUI test scripts developed by the Android development team of the SUT.

TESTAR was run with 15 different configurations. Three exploration ASMs were supported by TESTAR: 1) *Random* where an action is chosen from the derived list at random, 2) *Unvisited action first* where the state model is leveraged to select an action that has not previously been executed [189], and 3) the *Reinforcement Learning* algorithm *Q-learning (QL)* that rewards GUI exploration [80] by giving higher rewards to actions that have been executed fewer times. Then, five combinations of *number of test sequences (Seqs)* and *number of actions per test sequence (Acs)* that all amount to 3000 actions (i.e., 6 sequences of 500 actions, 10 sequences of 300 actions, 30 sequences of 100 actions, 50 sequences of 60 actions, and 100 sequences of 30 actions.)

Stoat and DroidBot do not work with test sequences. Both tools test until a limit for the total number of actions is reached. To achieve a fair comparison between TESTAR, Stoat, and DroidBot, the number of actions for Stoat and DroidBot is also set to 3000. For TESTAR, Stoat and DroidBot, a login sequence and the filtering of widgets are added as domain-specific knowledge. To ensure Droidbot did not escape the SUT, it had additional predefined actions specified.

Espresso test cases were the existing scripted test automation for the mobile application of the ING, developed by their Android developers. As Espresso tests are scripted, no settings can be modified. Although it is not possible to provide further details about the Espresso test scripts for confidentiality reasons, it is possible to reveal that these are the actual set of test cases that ING uses to test the SUT.

Dependent variables

Test effectiveness is measured through code coverage - Instruction Coverage (IC), Line Coverage (LC), and Method Coverage (MC) - using JaCoCo [127]. COSMO [244] is used to enable the integration of JaCoCo with scriptless dynamically created test sequences.

Design of the experiment

All experiments have been carried out on an Android emulator running on the same device (MacBook Pro, 8-Core Intel Core i9, 2,3 GHz, 8 cores) to ensure the hardware does not affect the test results. Due to the randomness involved in the ASMs, the coverage results can differ between runs with the same settings [19], and thus, runs should be repeated many times for statistical significance. However, due to the time restrictions, it is infeasible to run each TESTAR setting combination. The average run time was measured to be 287 minutes, meaning that 15 different settings would require approximately 90 days of continuous running. Therefore, only the best-performing TESTAR settings were executed ten additional times.

7.4.4 Results

The results can be found in Table 7.2. The best performing setting for TESTAR was Q-learning, with 10 sequences of 300 actions reaching a code coverage between 40.8-41.9%. This code coverage is slightly worse than the performance achieved by Espresso scripted tests. However, TESTAR achieves significantly better performance compared to Stoa and DroidBot.

As indicated, the best-performing TESTAR settings were executed another 10 times for statistical significance. We found that the average of ten runs is slightly lower (IC 40.4%, LC 40.3%, MC 40.7%) than the original coverage recorded. Additionally, the variance observed is low (IC 0.29, LC 0.34, MC 0.28), indicating the randomness in the QL algorithm does not have a big impact on the overall coverage results. Together, it gives us more confidence that TESTAR QL achieves performance that is very close to the Espresso tests.

As the performance of the Espresso tests and TESTAR are quite similar, it is interesting to examine the performance of TESTAR versus Espresso in more detail. To obtain a more detailed comparison, we investigate the code coverage at the package level. For each package, we recorded the code coverage of TESTAR and subtracted the code coverage of the Espresso tests. We found some packages showing a considerable difference between TESTAR and Espresso. The top 20% of the packages showing the most code coverage difference is displayed in Figure 7.7. The names of the packages are anonymized as they might contain confidential information.

TESTAR achieved better code coverage for the packages containing code that implements operations and settings for the end users to modify and customize the information shown for them in the SUT. These packages contain the code relevant for customers to modify what is visible but also the code relevant to displaying the actual information to the customer.

Tool	Alg	Seqs	Acs	% IC	% LC	% MC
TESTAR	UAF	10	300	34.9%	34.5%	35.8%
TESTAR	UAF	30	100	37.2%	36.8%	38.0%
TESTAR	UAF	100	30	37.0%	37.0%	38.5%
TESTAR	UAF	60	50	34.2%	34.0%	35.4%
TESTAR	UAF	6	500	35.2%	35.1%	36.7%
TESTAR	RND	10	300	40.2%	39.9%	40.2%
TESTAR	RND	6	500	36.7%	36.7%	37.7%
TESTAR	RND	30	100	36.2%	35.6%	37.0%
TESTAR	RND	100	30	35.6%	35.5%	37.1%
TESTAR	RND	60	50	38.1%	38.3%	39.3%
TESTAR	QL	10	300	41.0%	40.7%	40.8%
TESTAR	QL	30	100	37.7%	37.9%	38.8%
TESTAR	QL	100	30	39.1%	39.0%	40.0%
TESTAR	QL	60	50	38.4%	38.5%	39.5%
TESTAR	QL	6	500	40.4%	40.3%	40.8%
Droidbot	UAF	-	-	34.9%	34.1%	35.7%
Droidbot	RND	-	-	34.3%	33.7%	35.3%
Stoat	-	-	-	23.7%	22.6%	25.7%
Espresso scripted tests				43.9%	43.4%	45.9%
Combined TESTAR and Espresso				52.3%	52.1%	52.3%

Table 7.2 Coverage results for the Android application

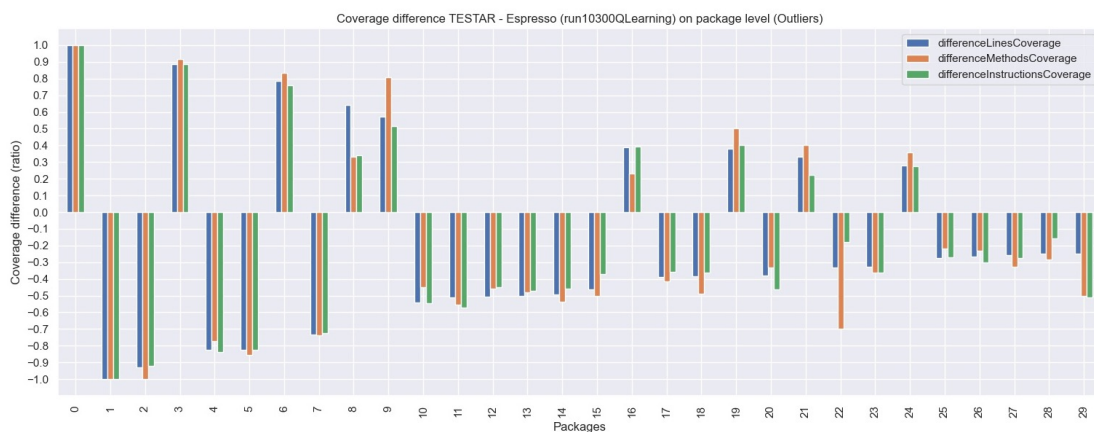


Fig. 7.7 Figure showing the top 20% of the packages with the most difference in coverage between TESTAR and Espresso

TESTAR is capable of covering this code as it is able to log-out from the application during the test sequence and continue exploring the SUT after that.

Another example of outlying packages where TESTAR outperforms Espresso in code coverage is related to exporting customers' own information and data from the application. Specifically, TESTAR covers the code related to generating PDFs with different types of information. Depending on the type of information to be exported, a different PDF-generating code of the SUT is called. TESTAR seems to be capable of performing the export function from multiple contexts.

As the performance of the Espresso tests is slightly better for the complete SUT, there are numerous packages where Espresso outperforms TESTAR. Espresso achieves greater code coverage for all packages related to security. A number of the security packages are in the outliers of Figure 7.7. Additionally, the packages for login obtain higher code coverage through the Espresso tests. TESTAR has a predefined login sequence that is triggered in the login screen of the SUT. Login was defined as always using valid credentials. This could be the cause for the lower code coverage achieved in the login packages. Adding more SUT-specific information about the login procedure to TESTAR, for example, adding the possibility of using wrong credentials, might increase the coverage of the scriptless approach for these security packages.

Due to the difference in coverage between packages, it is interesting to determine what the code coverage is when Espresso and TESTAR are used together. Using JaCoCo, we obtain the following code coverage for Espresso and TESTAR together: IC 52.3%, LC 52.1%, MC 52.3%. This shows a significant increase in coverage. Moreover, we see that the scriptless approach covers parts of the application under test that the existing test scripts did not cover, *showing the complementarity of the approaches*.

In summary, TESTAR outperforms both Stoa and DroidBot in terms of code coverage. When comparing TESTAR to the Espresso test scripts, the results indicate that scriptless testing complements test scripts, as TESTAR and the Espresso scripts covered different parts of the code. Consequently, the combined coverage of both approaches was higher than with either approach alone. Due to the type of the SUT, the end users expect that the application can be trusted to work flawlessly in all cases. Therefore, also the paths outside the main user scenarios are important to be covered during testing.

Threats to Validity

We discuss some threats to the validity according to [297, 19, 232].

Construct validity Code coverage was applied as a metric to evaluate the effectiveness of the testing methods. The code coverage does not directly relate to the ability of the

testing tool to find faults within the software, but it is generally accepted for measuring the performance of tests.

Internal validity We evaluated which components of the code base should be covered by the tests together with the Android developers of the ING. This ensures the code coverage is only measured for the packages the tool should be testing. Additionally, JaCoCo is verified by manually comparing the results for identical test sequences and checking the code coverage is identical.

External validity Although the SUT is an industrial application actively used by millions of people, it is only a single application. Additional studies are required to show that TESTAR works for other mobile applications too.

Reliability For most of the evaluated settings of TESTAR, the tool was only executed for a few times. As the algorithms used for exploration involve randomness, the results will vary between the runs. Due to the long execution times, it was infeasible to repeat the execution of all the settings sufficiently to get statistically significant results. Therefore, the reliability threat could not be completely mitigated. However, the best-performing settings of TESTAR were executed an additional ten times, showing low variance. This indicates that the recorded performance can consistently be achieved, reducing the reliability threat.

7.5 Keen Software House & GoodAI industrial experience in 2023

In the early 2020s, the video game industry surpassed 2 billion players worldwide, generating an impressive revenue of 120 billion dollars. This remarkable trend is anticipated to experience substantial growth in the future [64].

Game testing in the industry predominantly relies on human game testers, who invest significant manual effort and time verifying that user interactions within virtual scenarios yield the intended outcomes [221]. However, as industrial games grow in complexity, companies face the inherent limitations of game testers' efforts.

Sandbox 3D games, such as the industrial game Space Engineers developed by Keen Software House and GoodAI companies, emphasize the freedom and creativity of users in virtual scenarios. Players are given a wide range of tools and resources to shape the game scenarios according to their preferences and playstyle. The testing team of Space Engineers comprises ten game testers who excel in assessing functionality to create, destroy, modify, or interact with in-game objects, verify visual aspects, and manage game scenarios. Nevertheless, despite the testers are dedicated to performing numerous daily manual tests,

the extensive range of in-game elements constrains their time for exploring and testing unforeseen scenarios.

Scriptless testing techniques do not rely on explicit test case instructions. Instead, they employ algorithms to dynamically generate non-sequential actions during run-time, enabling them to explore and discover SUT objects autonomously. This introduces randomness and variability, which helps to complement traditional testing by uncovering unexpected issues and performing unanticipated combinations of interactions [128, 40].

Encouraged by the spatial exploratory effectiveness results that the scriptless *agent* demonstrated with the experimental 3D game LabRecruits, Keen Software House and GoodAI decided to study the integration of the scriptless *agent* capabilities with the 3D sandbox game Space Engineers.

7.5.1 Space Engineers game SUT and testing process

Space Engineers¹⁰ is a sandbox 3D game developed by Keen Software House and GoodAI companies. The game is coded in C#, started its alpha release in 2013, transitioned to beta in 2016, and was officially released in 2019. In 2022, Space Engineers had an average of 5,000 players with peaks of more than 9,000 concurrent players. Between 2013 and mid-2023, the game has evolved over nearly 596 game build changes. The game is available on Steam and console platforms and has sold around 5 million units.

The Space Engineers game simulates realistic open-world 3D scenarios. Due to the nature of the sandbox game, there is no specific objective to finish the game. Users can explore planets in space, portray their idealistic spatial constructions, play challenging scenarios to survive, or collaborate and compete with other players.

Space Engineers game mechanics

In Space Engineers, all game objects reside in a position and orientation of a three-dimensional world and have properties that indicate the object name, velocity, and unique identifier inside the game environment.

The astronaut is a playable game character that allows users to be part of and interact with the game environment. The astronaut has various characteristics such as energy, hydrogen, oxygen, and health, and capabilities like flying using the jet-pack. Due to the game recreating an open space world, the astronaut can move, fly, and rotate in 3D scenarios.

The game has atomic *Block* objects with properties representing attributes like type, integrity, volumetric physics, mass, inertia, and velocity. These blocks can be categorized

¹⁰<https://www.spaceengineersgame.com/>

into *functional* and *structural* blocks. *Functional* blocks have the capability of executing a task. For instance, this functional task can be producing energy for power blocks or restoring the character characteristics for life support blocks such as a medical room block. *Structural* blocks do not execute a task on their own but are used to build constructions. For example, armor structural blocks are used to build the floor and walls of space stations.

Constructions are known as *Grid* objects. A *grid* can be as simple as a set of structural blocks that constitute the floor of a space station or a complex engine that extends the task capabilities of functional blocks. For example, a medical room that restores the astronaut's health can be connected with an O₂/H₂ generator to additionally restore the astronaut's oxygen.

To construct blocks or sustain functional tasks, the astronaut needs so-called *Items* like: *Tools* used to interact with blocks and game mines; *Ores* mined from planets or asteroids using drill tools; *Materials* refined from ores into useful ingots; *Components* crafted from materials and required to construct blocks.

Figure 7.8 shows a Space Engineers scenario with a functional medical room connected to a functional O₂/H₂ generator via structural conveyor blocks. The O₂/H₂ generator can refine ice ores and supply oxygen to the connected medical room. This allows the astronaut to restore oxygen when interacting with the medical panel. However, if the integrity of the O₂/H₂ generator is less than 80%, the ice ores cannot be refined, and the oxygen will not be supplied.

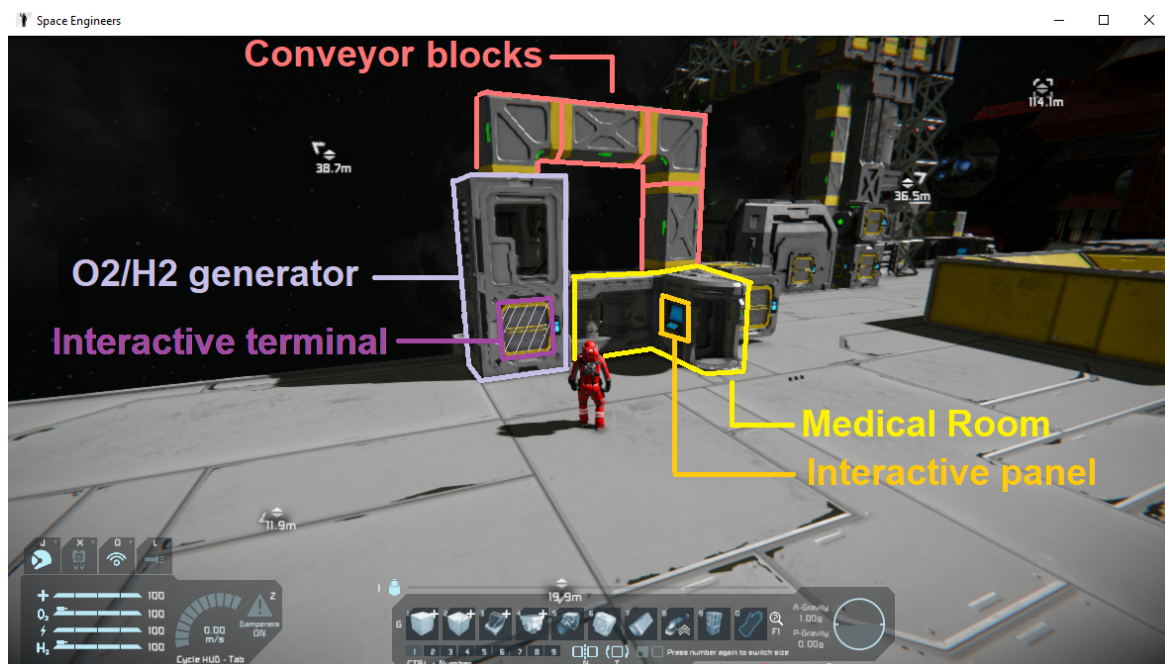


Fig. 7.8 Space Engineers game scenario

Space Engineers scenarios can be launched in *creative* and *survival* modes. *Creative* mode makes the astronaut invulnerable. His health, oxygen, hydrogen, and energy statistics will not decrease when resources are spent. Moreover, the astronaut can build blocks without the need to have the correct components in the inventory. In contrast, in *survival* mode, oxygen decreases over time, energy reduces with activity, hydrogen is spent when the jet-pack is used, and the astronaut can die if he/she loses its health points.

The variety of blocks and items, the diverse possible constructions to build with them, the scenarios game modes, and the 3D open-world movements make Space Engineers a highly complex game to test.

Development & manual testing cycle of Space Engineers

A typical Space Engineers game development cycle takes about 3 or 4 months, depending on the extent of game changes. This cycle involves two primary teams: developers and testers. The developers design and implement new game features and fix bugs reported by the testers during the release of new game versions. As developers finalize these changes, they open Jira tickets [87] to point testers to the features that require testing.

In turn, testers process developers' Jira tickets to verify the functionality of new or updated game features and validate potential bug fixes. Additionally, they assist the game community in verifying and documenting possible problems that users encounter when crafting specific scenarios. The testers team comprises ten members with different expertise roles that include: *console port* (testing features in console); *scene and world creation* (ensuring scenarios can be created, saved, and loaded); *surround sound* (checking sound volume in scenarios); *player support* (simplifying bug reproduction steps reported by community users to assist developers in resolving issues).

Testers' proficiency in understanding game mechanics from players' perspectives is crucial to accurately reproduce scenarios during sandbox game feature validation. As a result, manual testing remains essential to ensure comprehensive game testing. However, testers lack sufficient time to manually test the extensive and diverse combinations of entity interactions in the game. Manually exploring blocks and items to reach spatial coverage can be prohibitively costly and time-consuming. In light of these challenges, it becomes relevant to investigate scriptless testing as a complementary automated testing solution.

7.5.2 Space Engineers study with the iv4XR framework

The development of the IV4XR Space Engineers-plugin¹¹ enables access to the internal data and functions of the game. In Space Engineers, the *agent* takes the role of the astronaut.

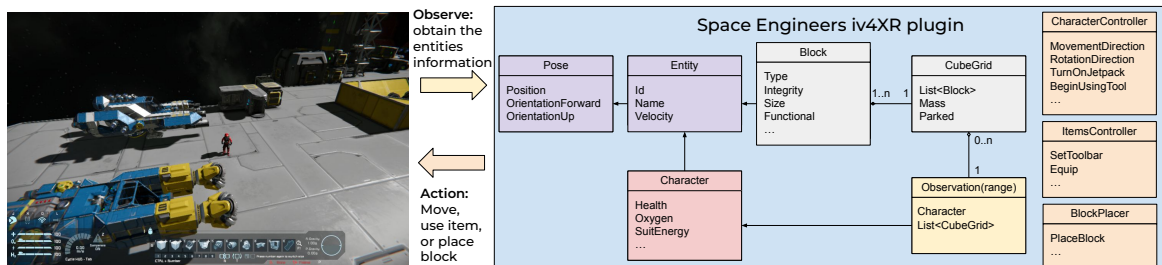


Fig. 7.9 Space Engineers-plugin overview

The Space Engineers-plugin consists of server and client components. The server-side is implemented in C#, has 8462 lines of code (LOC), and allows the connection with the game by defining the properties and controller functions of game objects. The client-side is written in Kotlin to ensure interoperability between the C# game and the Java IV4XR framework, has a size of 17671 LOCs, and provides classes that grant access to game data and logical functions like navigation for the *agents*. Figure 7.9 shows an overview of the plugin classes, which are discussed below.

Space Engineers-plugin entities

In the Space Engineers environment, each game object is represented as an *Entity* that stands in a specific *Pose*. The *Pose* denote their position and orientation within the game, and the *Entity* properties indicate each object's identifier, name, and velocity.

Block extends *Entity* with properties that indicate the block's type, integrity, and size, together with an attribute that indicates if the block is of the functional category (e.g., power block or medical room). *CubeGrid* contains the list of blocks that compose a grid (e.g., a spaceship grid is composed of a cockpit, thruster, and power blocks), and properties representing the grid mass and if the grid is parked (e.g., a spaceship is parked or is being controlled). The *Character* entity extends *Entity* properties with the astronaut's characteristics of health, oxygen, energy, etc.

¹¹<https://github.com/iv4xr-project/iv4xr-se-plugin>

Space Engineers-plugin observation

The *agent* connects with the *Character* through the Space Engineers-plugin to *Observe* the Space Engineers environment. The *Character* is always present. The existence of *CubeGrid* and *Block* entities depends on a configurable observation range of the *agent* and its distance from the game objects. Figure 7.10 shows how the observation range, a 3D sphere, works in the Space Engineer's environment. The *agent* observes itself, the main platform grid, and one spaceship grid. As we have explained, the grids are composed of a set of block entities. The spaceship grid is composed of a cockpit block, thruster block, power block, etc. The grid platform is composed of a group of structural *ArmorBlock* representing the floor of the scenario, together with functional *MedicalBlock* and *PowerBlock*.

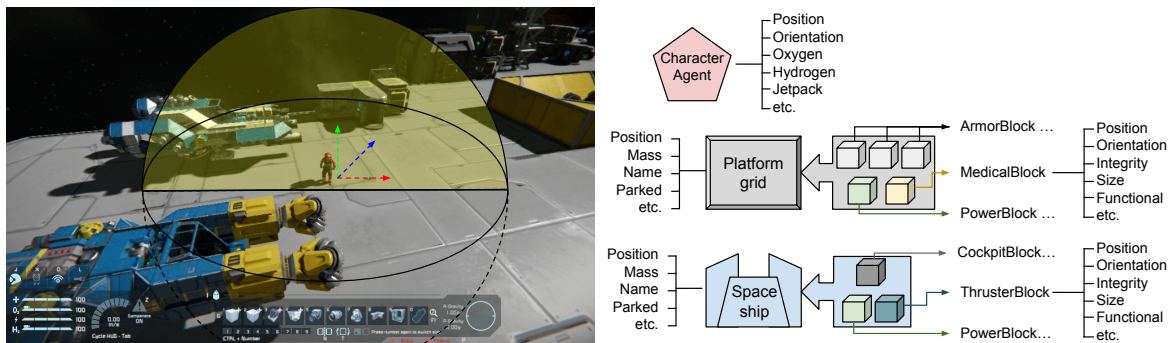


Fig. 7.10 Space Engineers observed environment by the agent

Space Engineers-plugin actions

The Space Engineers-plugin allows the *agent* to control the *Character* to interact with the game. The plugin controllers call internal game functions to move or rotate the character, turn on/off the jet-pack, equip/unequip an item, place a block, etc. To invoke these controls, the *agent* executes *commands*.

Observing the entity's data within the game environment and executing *commands* by invoking game controls make the Space Engineers-plugin a more robust approach compared to the usage of keyboard and mouse inputs or visual recognition tools that lack access to the internal game data. However, *commands* are insufficient even to accomplish simple tasks such as grinding a block. It is necessary to group sequences of *commands* in *actions*. For example, an *action* to grind a block is composed by the *commands*: Find the grinder tool, equip the grinder, aim the block, start using the grinder, and stop using the grinder.

Space Engineers-plugin navigation

Compared with Unity games, the initial version of the Space Engineers-plugin did not have the capability to create a default *navigation* mesh [226]. Instead, it constructed a *navigation* graph on-the-fly using the geometry information of observed entities. This approach has drawbacks, as it incurs a significant time cost after each game exploration movement and is not robust in three-dimensional space, where the *agent* could dynamically change its orientation.

To enhance game *navigation*, the Space Engineers team introduced automatic graph calculation for each *CubeGrid* entity. This calculation generates a list of positions the *agent* could reach without obstructions. In Figure 7.11, the *agent* observes the navigable positions, allowing him to create actions with a path of command movements to reach the interactive entities.

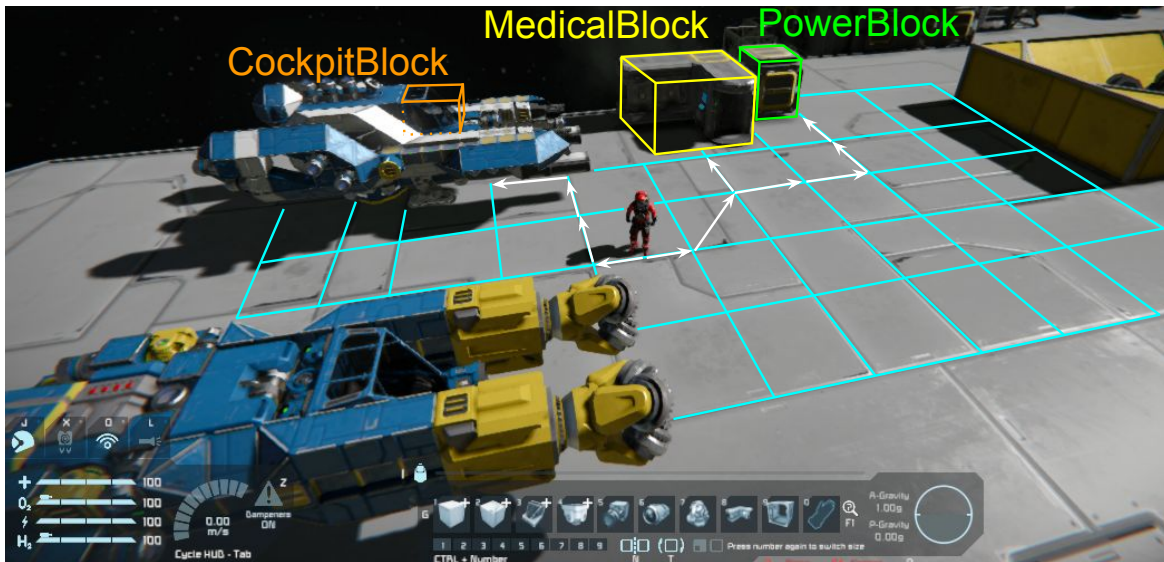


Fig. 7.11 Navigable actions in Space Engineers to reach interactive entities

7.5.3 Space Engineers study with the TESTAR-iv4XR agent

Using the Space Engineers-plugin, the TESTAR *agent* can perform the scriptless operational flow shown in Figure 7.12 to test the industrial 3D sandbox game.

TESTAR agent: Game state in Space Engineers

The TESTAR *agent* employs the Space Engineers-plugin to actively *Observe* all the game entities that reside in the observation range area. Each game entity contains a set of afore-

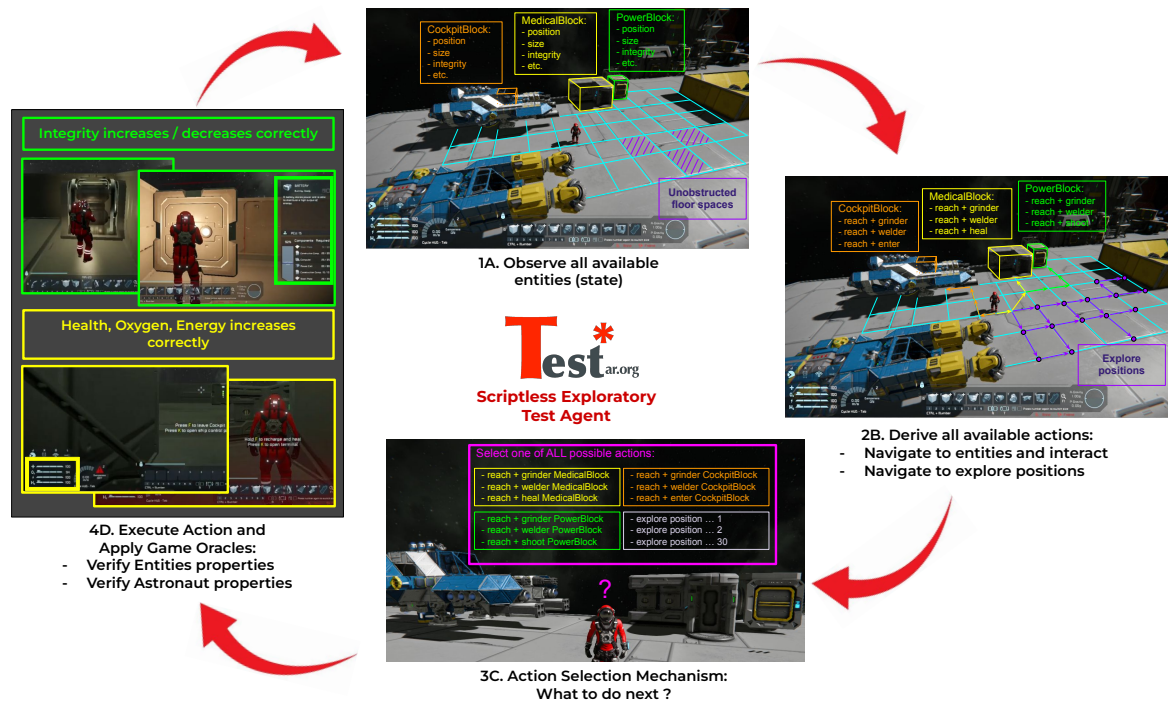


Fig. 7.12 TESTAR operational flow with Space Engineers

mentioned properties, such as position and orientation for all entities; health and oxygen for the *Character* entity; and type and integrity for *Block* entities. Together, these observed entities and their properties constitute the game *state*.

TESTAR agent: Derived actions and navigation in Space Engineers

Depending on the type and other entity properties, the TESTAR *agent* derives all the available actions that can be executed for each entity. For example, to grinder or welder all non-structural *ArmorBlock* entities, or to interact with *MedicalBlock* or *CockpitBlock* functional blocks to restore oxygen.

Similar to LabRecruits, the Space Engineers-plugin's *navigation* graph and the A* pathfinding algorithm provided by the IV4XR framework facilitate finding the optimal movement path between initial and destination nodes. However, it is necessary for a decision-making step at the top level to determine which action to derive and select during the exploration process.

To reach the desired *Block* to interact, the TESTAR *agent* exploits the *navigation* capabilities of the Space Engineers-plugin to observe the unobstructed floor spaces and calculates if there is a *navigable* path of positions that can be followed to reach the *Block*. If so, TESTAR derives an *action* that *navigates* the path, rotates to aim the *Block*, and interacts with the

Block using a *Tool*. However, if the *Block* is not reachable, TESTAR does not even try to derive an interaction with the *Block action*.

To potentially discover new game entities, the TESTAR *agent* not only considers deriving actions that interact with observed reachable *Blocks* but also derives actions that explore unobstructed positions. To do this, TESTAR's protocol has been extended so that after deriving all available interaction actions with unobstructed *Blocks*, it also derives all available exploration actions to unobstructed positions.

TESTAR agent: Action Selection Mechanism in Space Engineers

A random ASM will have the same issue when exploring the 3D sandbox game Space Engineers as it had when exploring the experimental game LabRecruits. Let us consider the example in Figure 7.12. First, the TESTAR *agent* observes 3 functional blocks (Cockpit, Medical, and Power) and derives 3 different actions for each block to navigate and interact with. This computes a total of 9 navigate and interact actions with functional blocks. Second, because there are 30 unobstructed positions in the observation area (e.g., imagine there are 30 purple dots), the TESTAR *agent* derives other 30 available exploration actions.

A random ASM will have less than 25% probability of selecting one of the 9 available interaction actions from the 39 total actions. This increases the chance of selecting an exploration action to more than 75%. Moreover, within the set of available exploration actions, selecting remote areas that remain unexplored can potentially allow the TESTAR *agent* the discovery of new entities. To enhance the exploration of unexplored areas, we have adapted the so-called **Space_Engineers_Interactive_Explorer_ASM**.

The `Space_Engineers_Interactive_Explorer_ASM` tracks a list of *interacted* entities and an area containing the *explored* positions. First, the ASM checks whether the set of available *actions* contains an action that interacts with a non-interacted entity (line 1). In that case, because there can be several non-interacted entities, it prioritizes choosing the nearest entity (*nearEntity*) to the *agent* (line 2). Thus, the ASM selects the action that navigates and interacts with the *nearEntity* (line 3), saves this *nearEntity* as interacted to not to be prioritized in the next iterations (line 4), and finally, returns the selected action (line 12).

Second, the ASM checks whether the set of available *actions* contains an action that explores a position out of the *explored* area (line 5). If so, because there can be several unexplored positions, it prioritizes choosing the remote position (*remotePos*) to the *agent* position (line 6). Consequently, the ASM selects the action that navigates and explores the *remotePos* (line 7) and includes the position in the *explored* area to enhance selecting other unexplored positions in the next iterations (line 8). Finally, the ASM returns the selected

Algorithm Space_Engineers_Interactive_Explorer_ASM

Require: <i>interacted</i>	▷ List the interacted entities
Require: <i>explored</i>	▷ Area of explored position
Require: <i>actions</i>	▷ All available state-actions

```

1: if actions contains entities that were not interacted then
2:   nearEntity ← nearestEntity(actions)
3:   a ← select to navigate and interact with the nearEntity
4:   save nearEntity as interacted
5: else if actions contains positions that were not explored then
6:   remotePos ← remotePosition(actions)
7:   a ← select to navigate to explore the remotePos
8:   save remotePos as explored
9: else
10:  a ← random selection from all actions
11: end if
12: return a

```

	▷ Return the selected action
--	------------------------------

action (line 12). In case the actions do not contain a non-interacted entity or non-explored position (line 9), the ASM selects (line 10) and returns an action randomly (line 12).

Different ASMs can be configured in the Java protocol of TESTAR. This way, the game testers can adjust the decision-making of the *agent* based on the requirements of different Space Engineers scenarios or testing objectives.

TESTAR agent: Oracles in Space Engineers

TESTAR integrates generic oracles intended to verify the robustness of the game system: detect if the process has crashed or hung or if the state elements, or debugging logs, contain suspicious exception messages. Although these generic oracles are a good way to start with automated scriptless testing, for Space Engineers, it is of paramount importance to test also the functional aspects of the game entities.

Examples of oracles can be to check that the integrity of all blocks decreases after grinding or shooting or increases after welding; that the *agent's* health, oxygen, hydrogen, and energy are restored when interacting with medical rooms or cockpits; or that the jet-pack and the dampeners are not enabled automatically without player activation after entering a cockpit, medical room, or interacting a ladder.

These oracles have been studied in [226]. In this paper, we apply oracles that validate the integrity of blocks, but we mainly emphasize evaluating the effectiveness of ASMs exploration.

7.5.4 Empirical evaluation

In order to assess the efficacy of scriptless testing for exploring the Space Engineers game, we evaluate the potential benefits of investing time and effort in developing ASMs for more sophisticated exploration techniques. To accomplish this, we quantitatively measure the spatial coverage of discovered and interacted entities and navigated positions within a randomly generated scenario. To guide our study, we have formulated a research question and null hypothesis:

RQ: How effective is spatial exploration in the Space Engineers game when using different TESTAR ASMs?

H₀: The Space_Engineers_Interactive_Explorer_ASM is not more effective than a random ASM in the Space Engineers game.

We designed a second controlled experiment based on Wohlin's guidelines [297] and a methodological framework specifically built to evaluate software testing techniques [278].

The experiment similarly consists of running the random default ASM and the more intelligent decision-making from Space_Engineers_Interactive_Explorer_ASM that prioritizes the interaction with newly discovered blocks and the exploration of remotely unexplored areas. Each trial measures spatial coverage of discovered Space Engineers blocks and floor positions.

Space Engineers generated scenario

A randomly generated Space Engineers scenario was used to ensure that the evaluated ASMs were not biased. The scenario consists of a 100x100 map with 8157 navigable positions and obstructive walls that TESTAR must navigate around to reach interactive blocks. The number of interactive blocks is randomly placed in a uniform distribution in various reachable parts of the map. From the 62 blocks specified by the company as fundamental for manual testing, we chose 16 types of 1x1 blocks that were allowed to be placed in the random scenario creation. Gravity blocks are also included to simulate gravity, resulting in 313 functional blocks of 17 different types.

Independent variables

To focus on the exploratory capabilities of the ASMs and prevent the *agent* from dying, we load the generated scenario in creative mode.

The TESTAR *agent* can use diverse tools and weapons to test the integrity of blocks. However, since this study focuses on spatial exploration, we applied the *blocking principle* [297] to limit the TESTAR *agent* interactions to a grinder tool that verifies that the integrity of

functional blocks decreases and shooting one bullet at gravity blocks to reduce their integrity without destroying their gravitational functionality. We also limited the observation range through the Space Engineers-plugin to encourage exploring and discovering new blocks.

Dependent variables

To answer our research question, we measured the number of discovered and interacted blocks, as well as the observed and walked positions. The Space Engineers game stores the scenario data in local XML files. This data contains information about the floor positions and existing blocks, and we compare it with real-time observations during the exploration process to obtain spatial coverage. Using this data, we can generate a 2D map highlighting the covered space.

Figure 7.13 shows an example of one exploratory sequence of 500 actions in the experimental map. Regarding navigable positions, yellow squares represent floor positions, red squares obstructive walls, blue circles observed areas, and green dots walked positions. Then, about interactive blocks, magenta dots represent not observed blocks, pink dots observed but not interacted blocks, and orange dots interacted blocks.

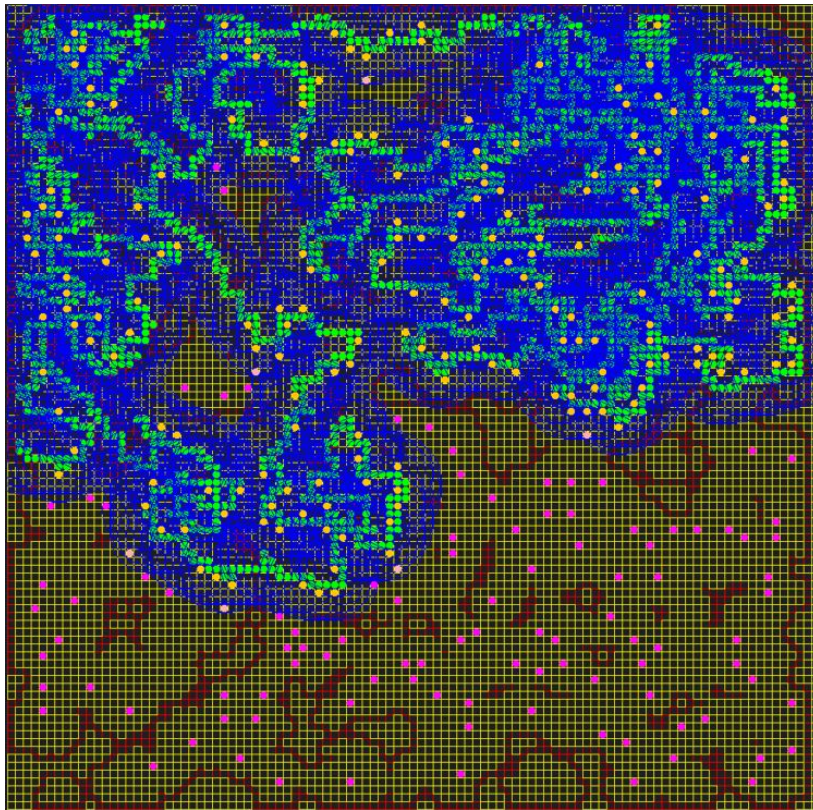


Fig. 7.13 Space Engineers spatial coverage map

Design of the experiment

We evaluate the random and the Space_Engineers_Interactive_Explorer_ASM by executing an exploration of 500 actions on the generated scenario. We repeated the exploration process 30 times for each ASM. For each new execution, we reload the same initial conditions in the same Windows machine with 8 CPU cores and 16 GB RAM. We obtain independent spatial coverage metrics for each execution and accumulative spatial coverage metrics for the 30 executions of the different random and Space_Engineers_Interactive_Explorer_ASM.

7.5.5 Results

We first present the spatial coverage achieved in the 30 independent runs. Next, we use the Wilcoxon test to determine whether there is a significant difference between the ASMs. The experiments were performed in Space Engineers v201.14. The replication package can be found here ¹².

Figure 7.14 shows the results for the observed and interacted blocks. Each line represents one of the 30 independent runs. The random ASM achieved a coverage ranging from 8% to 30% for observed blocks and 3% to 8% for total interacted blocks. In contrast, the Space_Engineers_Interactive_Explorer_ASM achieved coverage ranging from 54% to 82% for observed blocks and 52% to 77% for total interacted blocks.

Figure 7.15 shows the results for the observed and walked floor positions. The random ASM achieved a coverage ranging from 7% to 30% for observed positions and 4% to 12% for walked positions. In comparison, the Space_Engineers_Interactive_Explorer_ASM achieved a coverage ranging from 54% to 77% for observed positions and 21% to 24% for walked positions.

The Space_Engineers_Interactive_Explorer_ASM outperforms the random ASM by prioritizing interacting with newly observed blocks and calculating efficient routes to unexplored floor areas. Table 7.3 shows Wilcoxon test results to verify a significant difference between the two ASMs. We extracted values from the 30 different runs when executing 100, 300, and 500 actions. This means we calculate the significant difference in 3 different moments of the exploratory process. For the observed and interacted blocks and the observed and walked positions, the Wilcoxon test results show a p-value of less than 0.05, indicating that the Space_Engineers_Interactive_Explorer_ASM is statistically superior to the random ASM. This allows us to reject H_0 and confirm that investing time and effort in developing intelligent ASMs benefits TESTAR exploration effectiveness.

¹²<https://doi.org/10.5281/zenodo.10683676>

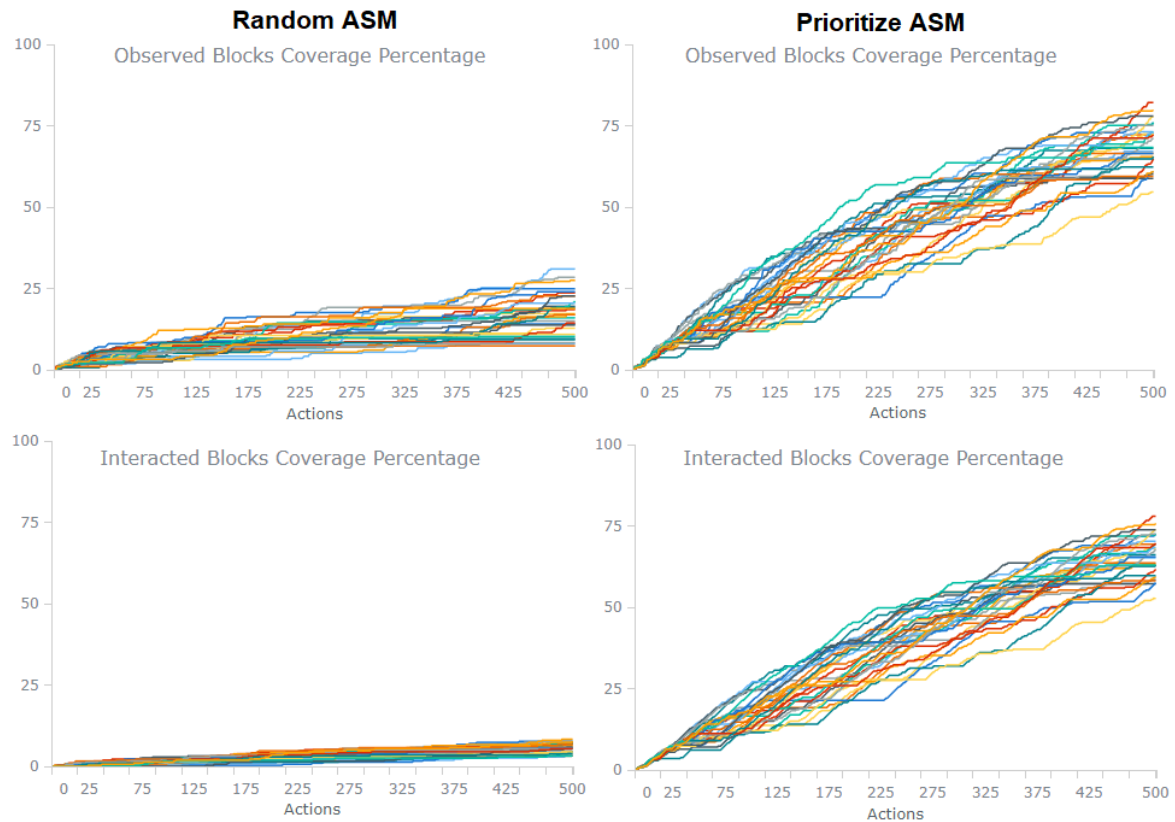


Fig. 7.14 Observed and interacted blocks coverage

Table 7.3 Wilcoxon p-value significant difference for Space Engineers

Wilcoxon test p-values results		
Executed actions	Observed Blocks	Interacted Blocks
100 actions	$p=1.730e-06$	$p=1.718e-06$
300 actions	$p=1.732e-06$	$p=1.734e-06$
500 actions	$p=1.730e-06$	$p=1.729e-06$
Executed actions	Observed Positions	Walked Positions
100 actions	$p=1.734e-06$	$p=1.733e-06$
300 actions	$p=1.734e-06$	$p=1.732e-06$
500 actions	$p=1.734e-06$	$p=1.734e-06$

Threats to validity

We discuss some threats to the validity of Space Engineers, according to [297, 232].

Construct validity For the exploratory evaluation, we use the information from the Space Engineers scenario to design the concept of spatial coverage. Then, we use this data to

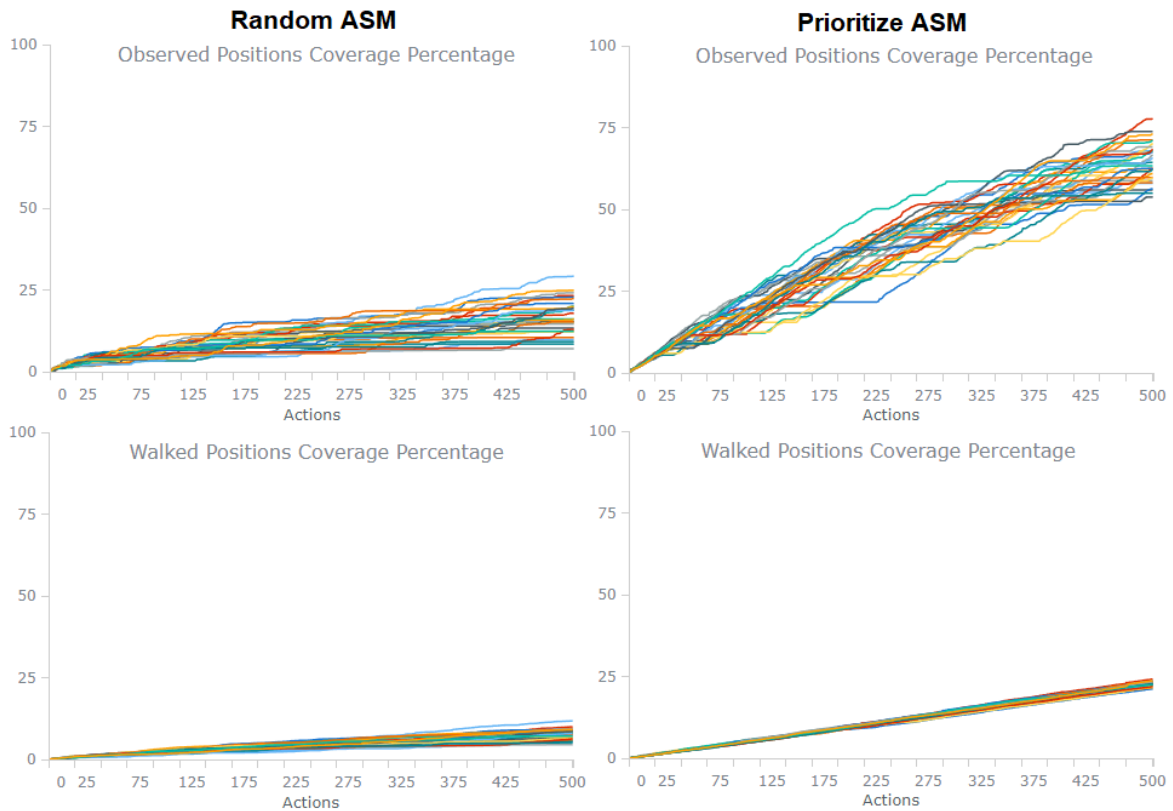


Fig. 7.15 Observed and walked floor positions coverage

measure the effectiveness of the random and Space_Engineers_Interactive_Explorer_ASM. Although this spatial coverage is a self-design benchmark, the metrics come from the Space Engineers game's data.

Content validity For the exploratory evaluation, the spatial coverage measures the existing blocks and floor positions over a 2D scenario space. Still, there are various types of blocks, and the game environment allows 3D motions. Although more sophisticated spatial coverage metrics can be researched in the future, the obtained 2D metrics allow us to measure the effectiveness of the exploratory ASMs. Moreover, while we did not encounter any bugs related to the integrity of the functional blocks used, it's important to emphasize that our solution effectively covered the scenario space. The lack of failure detection may be attributed to the random distribution of the test scenario or the absence of issues in the types of blocks utilized.

Internal validity For the exploratory evaluation, we launched the Space Engineers scenario in creative mode to avoid the astronaut dying and provide enough ammo items to realize the shoot gun actions.

External validity The empirical study has been realized with the highly complex Space Engineers game. Even though we demonstrated that the TESTAR *agent* has exploratory capabilities to navigate and test Space Engineers automatically, we consider this to be a first step regarding game scriptless test automation. Moreover, to facilitate the generalization of our results, we use the architectural analogy [293] since we carefully describe the components of the case and the corresponding interactions, such as the game system and the scriptless tool with the corresponding configuration.

Conclusion validity Due to the degree of randomness in the action selection of the exploratory ASMs, we cannot assume normal distribution in the experiments [18]. To address this, we repeated the exploration 30 times and used Wilcoxon statistical non-parametric tests on the results.

7.6 DigiOffice industrial experience in 2023

The context of this in-progress case study [43] is the software development company I.D.B. Telematica B.V., which is the vendor of the software solution DigiOffice¹³.

7.6.1 DigiOffice web SUT and testing process

DigiOffice has approximately 75 customers and approximately 26,000 end-users. Multiple customers have multi-million documents stored in their DigiOffice environment. Twice a year, a major version of DigiOffice is released. Ten times a year, a service pack is released for the latest two major versions.

Testing is done automatically with unit and integration scripted tests accompanied by manual human tests. There are 892 automated integration tests and 630 automated unit tests. The human testers have 213 manual GUI test plans for almost any application feature. The team uses TargetProcess as their agile planning and bug administration and SCRUM as their planning methodology. User stories and bugs have been recorded since 2008 in TargetProcess, and to date, approximately 13,000 user stories and approximately 12,100 bug reports in any development state.

The development team comprises 4 product managers, 10 senior developers, 2 senior testers, and 15 senior consultants. However, the development and testing teams are relatively small compared to the software product size. As a result, developers and testers cannot thoroughly test all documented user stories and regression bugs each time. This limitation forces the test team to make strategic decisions about what to test, how to test it, and when to

¹³<https://www.digioffice.nl/>

focus on specific parts of the system. Consequently, DigiOffice is exploring the capabilities of an automated scriptless testing tool that helps to complement its existing scripted and manual testing processes.

7.6.2 DigiOffice study with TESTAR

This study aims to determine the types and percentages of historical bugs in the DigiOffice database that could be detected using a scriptless testing tool like TESTAR. Additionally, it seeks to evaluate the effort required to create implicit test oracles that can be integrated into scriptless tools.

From the 12,100 bug reports available in the TargetProcess database, 2,500 were randomly selected for examination to ensure the study's statistical representativeness. The study was designed in two major steps based on this set of bug reports. First, an analysis was conducted on the bugs from the historical bug database of DigiOffice. Second, implicit actions and oracle concepts were developed and implemented in TESTAR to measure their theoretical and practical effectiveness.

A senior tester from DigiOffice [43] examined 8 potential classification schemes, analyzing a small sample of 30 recently fixed bugs. However, this analysis revealed that to classify the set of 2,500 bugs and the entire DigiOffice bug database, more flexibility and consistency were required in categorizing bugs by their symptoms.

As a result of this analysis, a new classification taxonomy was developed. A total of 76 bug classes were identified and grouped into 5 categories: Data, User Interface, Exception, Resource, and Invalid. The following list provides examples of bugs within these groups:

- **Data:** Items in a dropdown list have duplicated values, or the SUT lacks appropriate input validation for special characters.
- **User Interface:** Widgets are clashing and overlapping, images are blurry due to resolution issues, or text contains spelling errors.
- **Exception:** The SUT displays exceptions, such as a null reference exception or query exception, in the GUI or browser console.
- **Resource:** There are performance issues when loading SUT resources or incorrect Uniform Resource Identifiers (URIs).
- **Invalid:** Bugs arise due to invalid configurations of the SUT environment, or the detected bugs are not reproducible.

TESTAR's regular expression capabilities allow the customization of test oracles to detect *Exception* bugs by matching words or phrases such as "exception" or "error". Additionally, bugs like *Resource* issues with incorrect URIs can also be detected if they are reported as a web console message or an error message in the GUI. However, to detect other types of bugs related to *Data*, *User Interface*, or performance-related *Resource* bugs, it was necessary to enhance TESTAR with new action and test oracle capabilities.

7.6.3 Results

From the set of 2,500 bugs, this collaborative study generated 100 test oracle ideas, classified as easy, moderate, difficult, very difficult, or impossible. Failures related to suspicious messages that can be customized with regular expressions were classified as easy or moderate to configure in TESTAR, depending on whether the suspicious message could be triggered within a path of five actions or less. If a programmatic oracle needed to be implemented, the test oracle was classified as difficult or very difficult. However, this classification is subjective and depends on the degree of knowledge and understanding of what TESTAR is capable of and how it can be implemented. For example, implementing a test oracle to detect duplicates in a dropdown list might be considered difficult because it requires programming skills, but the implementation logic is straightforward. In contrast, implementing a test oracle to detect widget clashes might be considered very difficult because it requires both programming skills and algorithmic logic.

A total of 35 new action and oracle ideas were implemented, and the regular expressions were configured based on DigiOffice's bug requirements. Some examples, already mentioned in Chapter 5, include the following:

- Check if a dropdown list or a table contains duplicate items.
- Check if a dropdown list or table is unsorted. The sorting type does not matter since it could be numerical, alphanumerical, or logical.
- Check if a dropdown list or table contains too many items (e.g., 250+ items).
- Check if a dropdown list or a radio button contains only one item to be selected.
- Check if two leaf widgets (e.g., buttons or text areas) are overlapping.
- Check if a table does not contain any visible row element.
- Check if there is untranslated text or spelling faults.

- Check if naughty characters provoke internal exceptions due to invalid input validation.

Table 7.4 presents a summary of the implemented test oracles that discover new verified bugs in DigiOffice.

Test Oracle description	N° of new bugs found
Suspicious message oracle in the GUI	34
Detect suspicious message errors in the console of the browser	20
Long text as input to test the application for overflow exceptions	1
Detect that a dropdown list has a lot of items	1
Detect a radio button group with only one available option	1
Detect that duplicate rows in a grid are shown	1
Detect a panel or group without visible widgets	1
Detect untranslated text	1
Widget clash detection	1
Detect visual errors	2
XSS injection attack	1
Use accent (') SQL character in text fields for server error	3

Table 7.4 Implemented TESTAR test oracles that have detected new bugs in Digioffice

This study provided valuable insights into software failures and test oracles, leading to the implementation of new oracles within TESTAR that successfully detected dozens of previously unidentified faults. The findings from this collaboration can offer significant value to both academic researchers and industry professionals. For this reason, we expect to publish a detailed version of this industrial experience with DigiOffice in a future research paper.

7.7 Summary

This chapter presents a series of industry-academia collaborations that the TESTAR team has participated in during the development of this thesis. These collaborations have been of relevant importance for the evolution and improvement of the TESTAR tool to meet the needs of industrial companies and improve their testing processes. Almost all of these ideas, which were already described in the previous chapter as part of TESTAR, have been integrated into the tool.

Collaborations with companies like Ponsse and Prodevelop have led to significant advancements in the identification of multiprocess desktop GUI systems, the implementation of image recognition technologies for GUI interactions, and the adaptation of the tool for

seamless integration into CI environments. Additionally, these partnerships contributed to the visual and structural refinement of the test results produced by TESTAR.

Further collaborations with the banking companies KuveytTurk and ING motivated the integration of web and mobile GUI plugins into TESTAR. These efforts also demonstrated the tool's complementarity by detecting unexpected errors not covered by traditional manual and scripted testing processes and covering additional underlying SUT code. Moreover, a recent collaboration with Keen Software House and GoodAI provided the opportunity to explore the integration of the novel iv4XR framework to autonomously connect, identify, and explore an industrial 3D sandbox game.

Finally, the ongoing collaboration with I.D.B. Telematica B.V., which is focused on investigating and implementing a set of GUI test oracles for scriptless tools like TESTAR, has already led to the complementary detection of dozens of new bugs in real-world industrial applications.

Chapter 8

Beyond TESTAR: A BDD testing approach

In our previously presented academic-industry collaborations, we compared manual and scripted testing approaches with TESTAR's scriptless exploration. Our empirical studies demonstrated the importance of applying multiple testing techniques, as each technique offers complementary results in covering diverse parts of the SUT and identifying different types of failures. For example, our collaboration with ING highlighted how the Espresso scripted approach and the TESTAR scriptless approach covered different code methods in a mobile banking application, effectively complementing each other [128]. Additionally, a study conducted by a master's student working at E-Dynamics showcased how the Selenium scripted approach and the TESTAR scriptless approach detected different types of failures in the Yoobi web industrial SUT [40].

These collaborations help us to research and learn about other relevant testing approaches that were already included or desired to be integrated into companies' software testing cycles. For instance, in our collaboration with Keen Software House and GoodAI companies under the European research project iv4XR, the goal was to integrate intelligent agents capable of testing various technical aspects and functionalities of virtual reality environments. One of the project outcomes was the integration of TESTAR as an exploratory agent in the Space Engineers game. The Space Engineers team employs thousands of mandatory regression tests to run and evaluate before releasing new game versions. However, exploratory agents like TESTAR are more of a complementary solution and do not cover all game functionalities or aspects of these regression tests. For this reason, beyond the TESTAR scriptless approach, we also investigated other types of agents designed to automate these regression tests.

In this chapter, in Section 8.1, we describe the usage of the Behavior-Driven-Development approach to integrate an intelligent iv4XR agent intended to automate the execution and test oracles of a subset of the Space Engineers regression tests.

8.1 Behavior-Driven-Development game testing

Behavior-Driven Development (BDD) is a software development process that fosters collaboration and communication between developers and testers [259]. Although there is a scarcity of BDD research linked to industrial practices [38], various studies have shown its increasing popularity and widespread use in diverse systems [218, 24]. Recognizing the advantages of BDD, the Space Engineers team decided to embrace its adoption to streamline the regression test automation.

8.1.1 Space Engineers manual regression testing practices

During a Space Engineers *game release*, a development cycle begins with the team of developers working on changing the game to add, update, or fix features (Figure 8.1). As developers finalize these feature changes, they open Jira tickets¹ to point testers to the development branch that contains features that require testing. Testers then process these Jira tickets and *manually* test the newly implemented, updated, or fixed features. This development cycle takes about 3 or 4 months, depending on the required changes of the new game version.

Once developers have implemented all the required changes, testers assess whether the changed features have reached a stable state. If testers discover bugs in some of the changed features, they report the unstable features to the developers, who continue improving the development. After addressing any identified bug and confirming the stability of the development branch, testers proceed with executing a regression test suite.

This suite comprises 1322 tests, encompassing game features related to the astronaut's movements, attributes, items usage, interactions with blocks, as well as graphic or sound aspects. Testers manually execute the 1322 tests and document the results in TestRail², a management system that supports the organization of the testing process. Upon completing the regression test suite, testers analyze the results to determine if all tests passed without detecting any bugs. However, if any bugs are found during testing, testers assess each bug's severity.

¹<https://www.atlassian.com/software/jira>

²<https://www.gurock.com/testrail/>

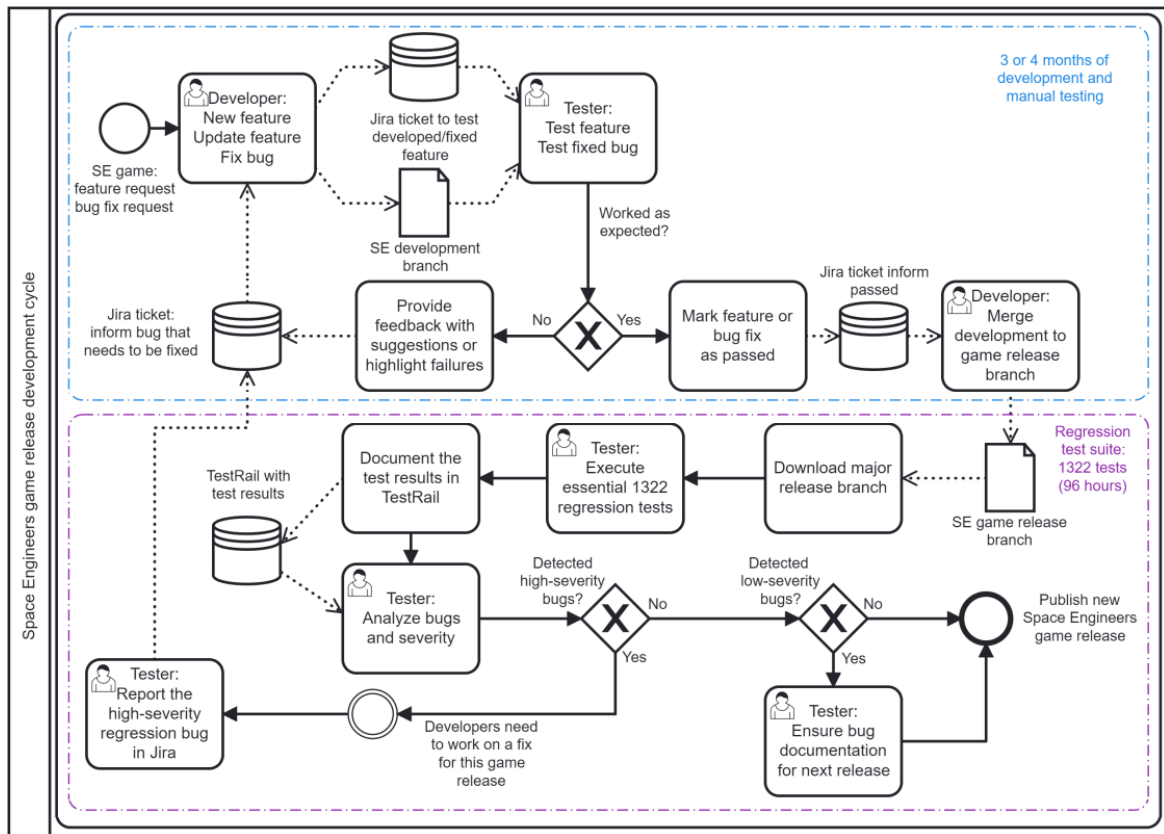


Fig. 8.1 Space Engineers development cycle diagram

If testers determine that a bug is of high severity, they open a Jira ticket, requiring developers to address it and fix the bug before proceeding with the game release. However, if the bug is classified as low-severity, the development branch can be published as a new *game release*, with the understanding that the low-severity bugs will be fixed in subsequent releases.

The regression testing phase described above is a time-consuming task that requires approximately 96 hours of manual effort. Due to the substantial manual time investment, integrating the regression test suite into the development cycle of a *game release* is not feasible. Instead, the regression testing is executed at the end of the cycle, typically occurring every 3 or 4 months. This delay in regression testing may result in possible bugs that could have been detected and fixed in the iterative development process but are now postponed for several months, potentially impacting the final steps of the *game release* process.

The proficiency and perception of human testers are essential to ensuring the accurate verification of astronaut, item, and block functionalities, as well as graphic and sound aspects. However, the Space Engineers team recognized the possibility of automating a subset of functional regression tests that are repetitive and time-consuming. By automating this

functional regression subset, nightly builds can seamlessly integrate into the development cycle. This integration will enable the early detection of possible regression bugs, significantly improving the overall effectiveness and efficiency of the *game release* process.

Challenges Automating Game-Testing Execution

The Space Engineers team evaluated the use of record-and-replay and visual recognition techniques to automate the execution of regression tests. However, they found a significant challenge as the recorded scripts proved to be unreliable in nearly all tests, even executed in the same scenario and game version. Reproducing the recorded astronaut movements and interactions did not yield the expected outcome, highlighting the lack of reliability in the recorded scripts.

Record-and-replay and other scripted GUI testing methods have shown fragility and unreliability, attributed to unpredictable GUI system behaviors and evolving GUI changes made between versions [111, 67]. Despite this, the GUI testing field has seen considerable progress. For example, accessing the web DOM data to generate reliable test scripts [200]. However, this progress has limitations for sandbox 3D games like Space Engineers due to the inability of existing tools to obtain the required game system information.

Automated tools for game systems rely on recording keyboard and mouse inputs, lacking access to internal game data such as position, orientation, and object properties. Thus, successfully reproducing recorded test actions becomes challenging, as the traversed states may exhibit slight variations in each replay. Unlike traditional GUI testing, this is not due to a test script being vulnerable to GUI modifications but rather because the test lacks the necessary resilience to account for the inherent dynamics and non-determinism of the game. To achieve effective game test automation, it is crucial to have access to the game object's positions for precise navigation and to the object's properties for test oracles.

Recognizing the constraints of record-and-replay tools in game automation and the need for an automation solution that connects with the game's internal data, the Space Engineers team opted to incorporate two key technologies. They chose the iv4XR framework [226] for creating reliable tests integrated with the game's internal data and the principles of Behavior-Driven Development (BDD) [259]. The latter enables game testers to actively design, generate, and maintain evolving test scenarios. We will discuss each of them in subsequent sections.

8.1.2 Test Scripts for regression test automation

The iv4XR framework supports the integration of an autonomous *agent* capable of connecting with and controlling a playable character, such as the astronaut in Space Engineers. This functionality is developed with a dedicated Space Engineers-plugin in Kotlin ³, enabling seamless integration and automated execution of regression tests. Figure 8.2 shows an overview of the plugin's architecture (A) and a Kotlin test script example that employs the plugin's interfaces (B).

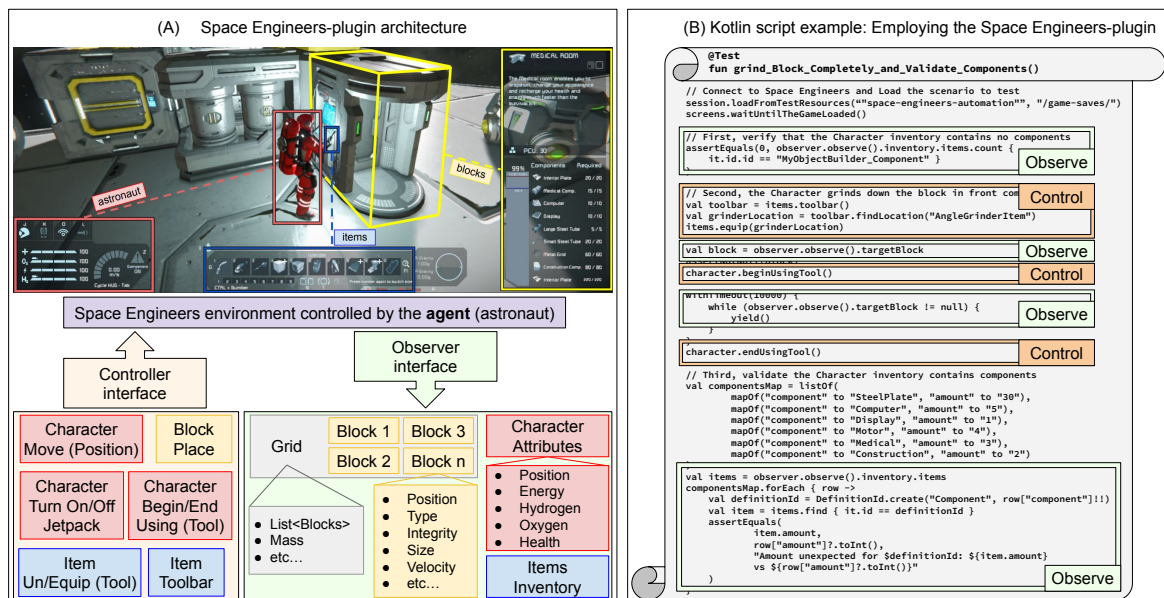


Fig. 8.2 Overview of the Space Engineers-plugin and Kotlin test script example

The *agent* gathers information about the surrounding game objects by *observing* the Space Engineers scenario. Since the astronaut character is always present, the *agent* can continuously monitor its own character attributes and inventory. However, block observation depends on the proximity of the *agent* to different grids and block entities. For instance, if the *agent* is near a construction, it will be able to observe all of the blocks, whereas being far away would not allow it to observe them. In the Kotlin test script example (B), observing the game objects involves identifying the inventory and the target block aimed by the *agent*. The inventory is observed initially to ensure it contains no items. At the end of the test script, it is observed again to validate that a specific amount of items can be found. On the other hand, observing the target block allows for checking its existence or whether it has been destroyed.

To ensure a reliable astronaut *agent* control, the Space Engineers-plugin incorporates *controller* classes that interact with internal game functions. These classes translate actions

³<https://github.com/iv4xr-project/iv4xr-se-plugin>

into executable game commands empowering the *agent* to maneuver the astronaut to move and rotate to specific vector positions, equip or unequip various items such as tools, utilize equipped items, and place equipped blocks. In the Kotlin script example (B), first, the items toolbar is controlled to find and equip the desired tool for the character. Once the tool is equipped, the character is controlled to begin using the tool until the target block is destroyed. When the target block is destroyed, the character ends up using the tool.

The iv4XR Space Engineers-plugin addresses the technical side and enables the creation of reliable test scripts to observe and manipulate the game environment through the iv4XR *agent*. This feature enables the automation of the regression test suite. However, it is necessary to create reusable scripts that foster collaboration between developers and testers. To facilitate this, we plan to employ the Behavior-Driven Development (BDD) software process.

8.1.3 BDD for regression game testing automation

To successfully implement the BDD process in industrial practice, it was essential to integrate a comprehensive suite of tools and techniques on top of the Space Engineers-plugin. This integration empowers testers to abstractly design and automate test script execution, freeing them from the need to delve into the technical intricacies of the plugin.

Given-When-Then theoretical structure

In a typical test scenario aimed at validating a game feature, the technical implementation involves the creation of a test script class with the following steps:

1. **Load Game Scenario:** The test script should programmatically load a specific game scenario containing the desired state objects and properties that require testing. This step sets the initial conditions for the test.
2. **Execute Actions:** The next step involves running a specific set of actions that will alter the desired game object. These actions simulate user interactions or in-game events that trigger changes in the system.
3. **Validate Results:** After executing the actions, the test script should validate that the game object and its properties have responded adequately to the actions. This verification ensures the game features work as intended.

To implement these steps in a Space Engineers test scenario, testers can create a test script in Kotlin utilizing specific functions from the Space Engineers-plugin. However, crafting

this Kotlin script requires technical knowledge of the existing plugin functions and valid game-plugin variables, which can pose challenges for non-technical testers in designing and maintaining test scripts.

The Given-When-Then (GWT) structure [201] is a standard format widely used in the BDD process. It enables the abstraction and description of system component functionality and expected behavior in a human-readable language:

- *Given* statement defines the Space Engineers level scenario and the game state that the test aims to validate. For instance, it can involve loading a specific level with the desired block to be tested.
- *When* statement enables to control the astronaut and execute a predefined set of actions that alter the Space Engineer's game objects. For example, the test instructs the astronaut to equip a grinder tool and destroy a block.
- *Then* statement verifies that the game object's properties have changed as expected, according to the specifications of the game feature being tested. For instance, it could check if the astronaut's inventory contains a specific amount of components after the block is destroyed.

Figure 8.3 illustrates how the Space Engineers test script written in Kotlin can be structured in a GWT structure. By adopting this structure in Space Engineers, testers can design and maintain regression test scenarios straightforwardly.

BDD in practice with Cucumber

Cucumber [74] is an open-source tool that facilitates the practical implementation of the GWT structure in the context of a BDD process. This tool has been integrated as a bridge between human-readable GWT statements and the technical Space Engineers-plugin functions that interact with the game.

During the implementation with Cucumber, Kotlin functions are created to encapsulate the necessary invocations, aligning with the objectives specified in the *Given*, *When*, or *Then* statements. Figure 8.4 shows how the original Kotlin script can be disassembled into functions to load game scenarios, execute actions, or validate game properties. Each corresponds to a specific GWT statement.

Once these functions are implemented, they can be seamlessly aggregated and invoked using Cucumber's human-readable language. This approach offers Space Engineer's testers an intuitive way to create, maintain, and execute test scenarios within the Space Engineers game environment.

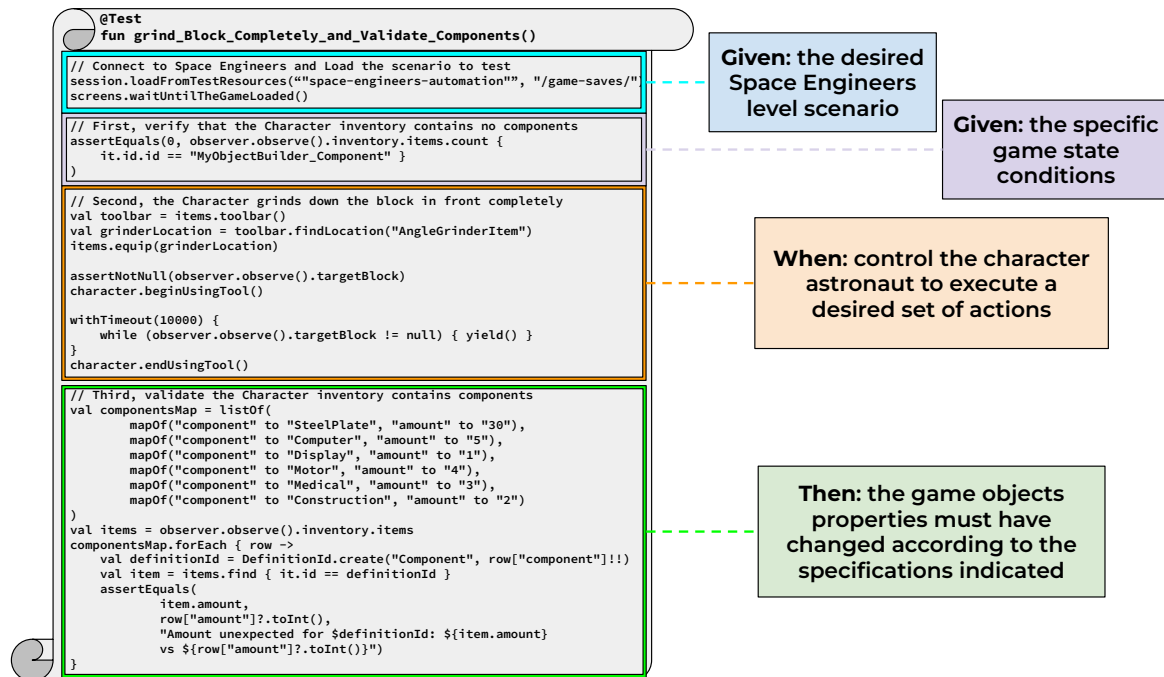


Fig. 8.3 Space Engineers Kotlin test script with GWT structure

In Figure 8.5, we present a human-readable test script resulting from the combination of the BDD process with Cucumber. This script represents a concrete Space Engineers regression test scenario that non-technical stakeholders can craft to validate the grinding of a block to collect components by the astronaut. In the subsequent section, we delve into the association between these test scenarios and specific game objects, such as T254794, providing a comprehensive understanding of how these scenarios are applied in practical testing situations.

Space Engineers pre-designed game level for testing

Executing BDD regression tests in a sandbox open-world game like Space Engineers poses an additional challenge. Our goal is to keep astronaut actions (*When*) and validation oracles (*Then*) as minimalist as possible. To do this, testers have pre-designed a game level (see Figure 8.6) that can be loaded using the *Given* statement. This level, divided into sections and stations, provides all the necessary blocks and items required by the astronaut in the diverse BDD test scripts.

Each section groups together generic features, such as astronaut movements, attributes, jet-pack, dampeners, and block grinding. In each section, multiple stations are set up with custom-arranged blocks designed to test specific functionalities. For example, a section is dedicated to validating the astronaut's health, oxygen, and energy attributes. In one medical

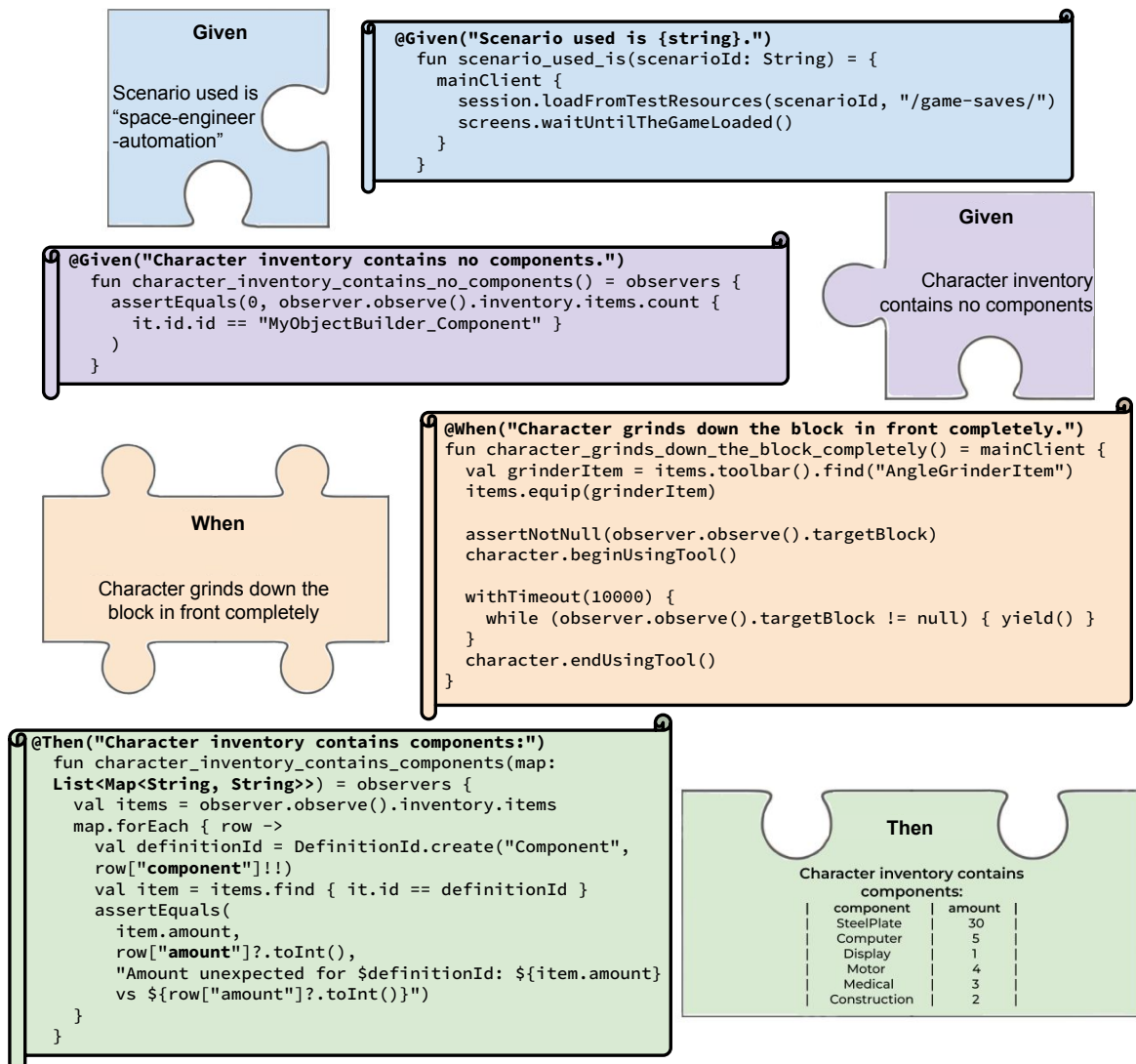


Fig. 8.4 Cucumber mapping of Space Engineers-plugin functions with GWT statements

station equipped with an oxygen tank, we validate the agent's oxygen replenishment. At another medical station without an oxygen tank, we validate that the agent cannot replenish his oxygen.

Each BDD test scenario is linked to the specific station that helps validate the intended functionality. To transit between test scenarios, the Space Engineers team utilizes a process that spawns the testing *agent* at the designated station.

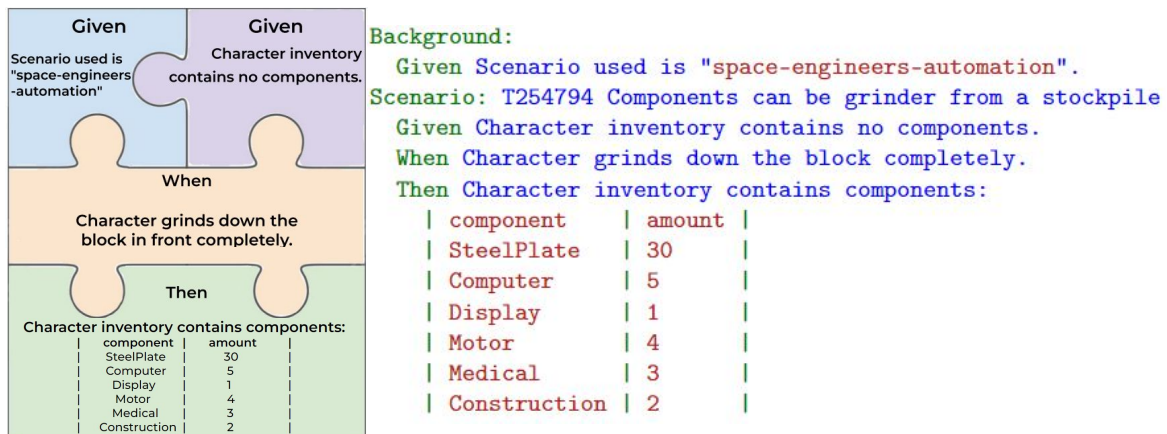


Fig. 8.5 Space Engineers BDD test scenario to validate astronaut inventory components

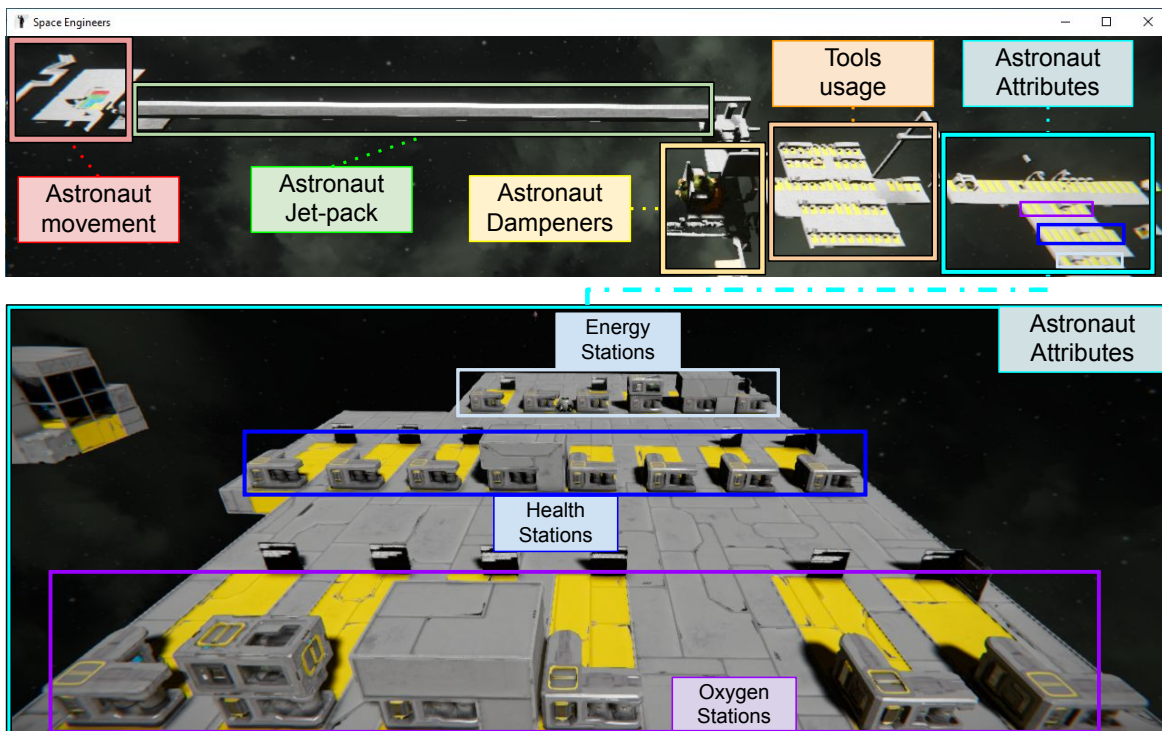


Fig. 8.6 Space Engineers pre-designed game level for testing

8.1.4 Industrial application of BDD test scenarios

The Space Engineers team decided to initiate the automation process with a subset of 236 tests from the total of 1322 regression tests for the following reasons:

1. **Essential Features:** The selected subset comprises essential features for validating the astronaut's movements and attributes. These functionalities play a significant role in ensuring an immersive player experience.

2. **Plugin Classes:** The current state of the plugin covers all the necessary functions required for automating the control of the astronaut's movement and interactions with blocks using tools and items.
3. **Oracle Validation:** It is feasible to integrate a reliable Oracle validation, which involves observing the state of the game, including assessments of the astronaut's position and attributes. Test scenarios that involved validating game graphics or sounds currently present challenges for validation.
4. **Manual Effort Time:** While executing each individual test is not time-consuming, the entire subset requires a considerable amount of time.

The Space Engineers team has successfully automated a subset of 236 regression tests using the BDD *agent*. In contrast to previous experiences, the BDD approach has demonstrated *reliability* in controlling the astronaut and applying validation oracles. This automation effort offers an opportunity for seamless integration of these 236 regression tests into nightly builds, marking an enhancement from the prior practice of conducting them solely at the end of the 3 to 4-month development cycle. Instead, the BDD *agent* can complement the daily verification work that testers do manually. The Space Engineers team estimates that this automation initiative could be equivalent to saving approximately 17 hours of manual effort each time the regression tests have to be executed.

Each BDD test scenario automatically generates a TestRail entry indicating whether the test passed or failed. If a test fails, the automated process generates a detailed report containing textual and visual information about the failed step. This seamless integration with TestRail has proven effective by successfully identifying a regression testing bug related to the engagement of the astronaut's magnetic boots. This success motivates further integration in nightly builds to streamline the regression test suite during the *game release* development cycle.

8.1.5 Threats to validity

This section presents threats that could affect the validity of our results [297, 232].

Content validity

While the number of successful automated regression tests can be quantitatively assessed, evaluating the satisfaction of the Space Engineers team is more subjective and difficult to measure from an analytical standpoint. In future work, we plan to continue automating

regression tests using the BDD *agent* and evaluate team satisfaction through qualitative feedback.

Internal validity

The selection of regression tests for automation relied on the expertise of the Space Engineers team. However, this experience could introduce selection bias and impact the representativeness of chosen regression tests. To mitigate this threat, the team fostered a collaborative environment where developers and testers with high expertise in the game led the selection process.

External validity

This industrial study with Space Engineers serves as a testing reference for a complex sandbox game. However, it is essential to acknowledge that games significantly vary in implementation and design. Therefore, to assess the generalizability of the iv4XR and BDD approaches, it is recommended to evaluate their effectiveness and applicability across a wider range of games.

Conclusion validity

The equivalence of saving 17 hours of manual effort resulting from automating the initial 236 regression tests provides a compelling advantage. However, it is important to acknowledge the absence of a controlled experiment, which limits the generalizability and statistical robustness of the conclusions. Future research should prioritize conducting other empirical evaluations to validate and reinforce the observed benefits of automated regression tests.

8.2 Summary

Regression tests involve the sequential execution of specific actions and integrating test oracles to check specific SUT conditions. Due to the inherent randomness involved in scriptless testing, tools like TESTAR complement these regression tests rather than serve as an automated testing solution. For this reason, this chapter describes the BDD testing solution we researched with Keen Software House and GoodAI companies beyond the scriptless approach offered by TESTAR.

The regression testing phase for the 3D sandbox game Space Engineers is a time-consuming task that comprises 1322 tests and requires approximately 96 hours of manual effort. This extensive time needed provokes the regression testing phase not to be feasible to

be integrated into the development cycle of a game release and be delayed at the end of the cycle, typically occurring every 3 or 4 months.

As a solution, a BDD approach has been integrated with the capabilities of the iv4XR framework to automate a subset of 236 Space Engineers regression tests. This automation effort translates into savings of approximately 17 hours of manual testing. Although further research is necessary to evaluate the maintenance and report analysis costs, the iv4XR-BDD approach presents a significant opportunity to seamlessly integrate these 236 regression tests into nightly builds during the development cycle.

Chapter 9

Discussion

In today's modern societies, software has become a fundamental pillar, encompassing nearly every aspect of daily life. From the digital economy and e-commerce to education, business, and entertainment, software systems have revolutionized the way people interact, work, and relate to each other. Moreover, software is integral to critical domains such as healthcare management, clinical support systems, energy supply networks, aviation, and banking. While the benefits of high-quality software in these sectors are immense—helping to improve the safety and overall quality of life of humans—there is also significant risk. When software fails to meet the indispensable quality standards, the consequences can be disastrous, especially in critical areas. In extreme cases, software malfunctions may even result in the loss of human lives.

This underscores the importance of software testing. Software testing is the primary means to ensure that a system meets its quality standards, encompassing diverse characteristics such as functionality, security, and usability. Traditionally, this task fell to human testers who would manually execute numerous tests to check the software's behavior. However, as software has grown in size and complexity, manual testing alone has proven insufficient. The evolution of technology has brought sophisticated software that demands more testing time and diverse techniques for assessing multiple software characteristics. Fortunately, technological advancements have also enabled the development of automated testing techniques, which help to complement and enhance human efforts by executing repetitive and time-consuming tasks or by exploring unexpected software states and paths.

Scriptless testing is one such advancement that helps to improve the quality assurance of software systems. Unlike traditional automated testing methods that rely on predefined scripts, scriptless testing dynamically generates test cases using exploratory algorithms. Often referred to as monkey testing, this approach performs non-sequential, random actions within the system, simulating unpredictable user behavior. While this method may appear

chaotic and useless, its simplest form, a completely randomness approach, may potentially offer a unique advantage: it can uncover paths and system states that scripted tests or even human testers might ignore. Scriptless testing serves as a powerful complement to manual and script-based techniques, testing the software in ways that were previously unconsidered. Nonetheless, it demands an improvement: improving its efficiency and effectiveness to avoid the need for long executions that do not align with the rapid and iterative software development demands of nowadays industrial companies.

In this thesis, we aim to further advance the field of scriptless testing with the goal of improving the quality assurance process for desktop, web, mobile, and eXtended Reality (XR) applications. Using the existing TESTAR tool as a foundation, we demonstrate how scriptless testing can evolve to enhance software quality assurance, covering unforeseen system characteristics and uncovering unexpected failures in complex, real-world industrial environments. To achieve this, we have made progress in four key TESTAR objectives: enhancing interaction with a diverse range of software systems, expanding its exploration capabilities, integrating effective test oracles, and ensuring its practical applicability with real-world industrial environments.

The technical and theoretical research conducted throughout this thesis, addressing various aspects of the scriptless testing approach and the challenges encountered, offers valuable insights for both academic researchers and industry practitioners. Solutions to these challenges, or their identification for future work, have implications for helping to advance in the test automation area. In the following subsections, we delve into the key TESTAR features of this research, highlighting the significance of the results and their potential impact on the testing landscape.

9.1 Interaction with Graphical User Interface systems

The software interaction chapter of this thesis presents how an automated testing tool can be extended with various plugins to connect to and interact with different Graphical User Interface (GUI) systems. GUI environments for desktop, web, and mobile applications exhibit diverse native characteristics, and the underlying technologies used to build these applications can differ significantly, even within the same operating system. As a result, GUI widgets expose their properties through distinct names and identifiers. However, despite these differences, the way widgets compose the GUI state and expose their properties share common similarities.

To address the challenge of integrating diverse plugin technologies, TESTAR implements a Taggable interface, allowing the tool to set or get property values of taggable widgets,

states, or actions using specific `API-Taggable` classes. This approach provides a flexible solution that integrates plugins for diverse GUI technologies into TESTAR. Each plugin manages the necessary internal operations to launch, connect to, and interact with a System Under Test (SUT), while the intermediate `API-Taggable` classes enable TESTAR to independently understand which are the widgets, states, and actions objects present in the GUI and their associated properties and values.

During this thesis, three new GUI technologies have been integrated into TESTAR: the Java Access Bridge, Selenium WebDriver, and Appium plugins. These technologies allow for seamless connection and interaction with desktop Java Swing, web, and Android/iOS applications. Additionally, the Windows UI Automation Accessibility framework plugin was evaluated and extended, improving its ability to handle multi-process desktop applications.

Although these are well-known plugins, these contributions may offer valuable insights for beginners in the test automation field. The plugins integrated into TESTAR facilitate automated interaction with a wide range of GUI software applications, making them applicable to various testing approaches. However, relying on multiple technical APIs and frameworks introduces challenges and further research solutions that must be taken into consideration.

Technical integration and maintenance effort

Integrating and maintaining multiple plugins within a testing tool like TESTAR presents a set of challenges. Each plugin relies on specific technical dependencies and libraries, which may have different update cycles, deprecations, and compatibility issues. Ensuring that the plugins remain functional and compatible with evolving software environments requires expertise and continuous effort. For instance, GUI libraries and frameworks, such as Selenium WebDriver or Appium, regularly release new versions that might introduce breaking changes or new features that need to be updated. Additionally, operating systems and application platforms undergo significant changes over time, which can demand extensive refactoring of the plugins. For instance, the transition from Windows 7 to Windows 10 introduced numerous changes to the Windows plugin, requiring adjustments to ensure compatibility. Likewise, Unix-based systems, such as Ubuntu, Debian, and Fedora, have varying implementations, each of which may have different requirements for interacting with GUI elements, further challenging the plugin integration efforts.

Another set of challenges involves managing plugin conflicts across multiple versions and implementing clean code to handle intrinsic technical differences between platforms. As each plugin is tied to specific environments or software technologies, updates to one plugin can affect the dependencies and functionality of TESTAR, leading to technical inconsistencies. For example, when the Selenium community dropped support for Java LTS version 8, changes

in internal dependencies—specifically in the ChromeDriver plugin—prevented TESTAR from maintaining Java 8 support without addressing these conflicts. As another example, Qt applications lack support for a Windows UIAutomation function that detects whether widgets are visible and ready for interaction. This limitation needs specific internal implementations in the technical plugin and the TESTAR core deriving actions.

Technical solutions can be implemented to address these challenges. For example, a tool like Dependabot, which automatically scans repositories for outdated dependencies and suggests updates, can help maintain plugin compatibility with the latest versions of libraries and APIs. Additionally, setting up Continuous Integration (CI) pipelines with automated tests for each plugin can help catch compatibility issues when dependencies are updated. Modularizing the plugins as independent components within the project can also help isolate issues and reduce the risk of cross-plugin interference. Finally, regular refactoring and documentation practices are necessary to keep the internal code of the tool clean and easy to maintain.

Using neural networks for GUI widgets recognition

The use of technical plugins is a robust approach for obtaining information about the GUI widgets that compose the SUT state. However, integrating and maintaining these plugins in scriptless testing tools can be challenging due to the diversity of GUI systems (e.g., desktop applications developed with .NET technology require different plugins compared to those developed with Java Swing) and operating systems (e.g., Windows, macOS, or various Linux distributions each require different plugins for desktop applications). Furthermore, some frameworks used to develop or customize GUI applications lack the necessary technical APIs or automation frameworks capable of detecting and extracting the properties of the widgets that compose the SUT state^{1 2}.

Object detection, a field of research that leverages Convolutional Neural Networks (CNNs) and other variants of Deep Neural Networks (DNNs), involves training models to identify objects within images [307, 299]. Additionally, Optical Character Recognition (OCR) is a related area that employs adaptive classifiers to extract text characters from images [260]. By combining existing CNN tools, such as Ultralytics YOLO³, with OCR tools like Tesseract⁴, scriptless tools like TESTAR can address the limitations posed by technical plugins that are insufficient for recognizing the properties of GUI widgets across specific SUT technologies or operating system environments [132].

¹https://github.com/TESTARtool/TESTAR_dev/issues/51

²https://github.com/TESTARtool/TESTAR_dev/issues/200

³<https://www.ultralytics.com/yolo>

⁴<https://tesseract-ocr.github.io/>

Text input complexity for type action

Click and slide actions may present abstraction challenges, particularly modifying widgets that introduce dynamic behavior. However, the interaction space for these actions is relatively straightforward, as they primarily involve mouse movements and clicks. In contrast, typing actions are inherently more complex due to the significant impact that typed text can have on the SUT and the vast combinatorial possibilities associated with text input. These combinations include a wide range of inputs—alphanumeric characters, emails, special characters, dates, varying lengths, valid and invalid data, security injections, and more—making text input a state-explosion challenge. Additionally, different SUTs or domain-specific contexts may require testing with unique datasets, further complicating the process.

The internal mechanisms of TESTAR automatically generate simple alphanumeric inputs, including emails and both valid and invalid dates. However, relying on these basic text inputs can limit test coverage and fault detection effectiveness. To address this, TESTAR has been extended to allow users to configure inputs manually through an external file, tailoring them to the specific needs of their SUT. For instance, this extension enabled users to configure security injection inputs designed to test and detect security vulnerabilities.

Despite this configuration possibility, input validation remains a significant challenge requiring further research. Fuzzing, a well-established technique that generates large volumes of unexpected input combinations to detect software bugs, can be explored as a potential solution [58, 164]. Nevertheless, the fuzzing technique still struggles to generate inputs focusing on SUT-specific or domain-specific behaviors. To address this, some TESTAR research solutions aim to instrument the SUT's code at runtime, extracting relevant string literals from executed statements in the source code to derive more targeted inputs [261].

Another challenge associated with text inputs lies in abstraction. By default, TESTAR ignores text when calculating abstract action identifiers to avoid the state-explosion problem. However, different text inputs can lead to distinct state transitions (e.g., submitting valid vs invalid data in a form). This abstraction approach, combined with the random generation of text input, can introduce non-determinism in the model inference process. In some of our thesis experiments, we programmatically designed compound actions that use specific valid or invalid text, allowing these actions to be treated as abstract and unique. However, this solution requires manual effort and in-depth knowledge of the SUT, making it a time-consuming process.

9.2 Interaction with eXtended Reality game systems

Compared to traditional GUI systems, eXtended Reality (XR) game systems require additional information to accurately identify objects and their properties within the game state. These include positional or orientation vectors for movements or properties associated with the objects being interacted with. Therefore, to enable scriptless or other automated testing methods, such as Behavior-Driven Development (BDD), this thesis addresses the technical challenges of establishing a connection between game test agents and integration technologies capable of interacting with game environments.

This thesis integrates the Intelligent Verification/Validation for Extended Reality Based Systems (iv4XR) framework, a novel Java-based solution for providing test automation for XR systems. Specifically, two game plugins developed by iv4XR partners enable the integration of scriptless and BDD test agents with the 3D games LabRecruits and Space Engineers. These contributions, along with the empirical evaluations, offer valuable insights for researchers interested in exploring the effectiveness of test agents in interacting with and testing 3D game environments.

Furthermore, Space Engineers, as a real-world industrial sandbox game, presents a valuable case study contribution for game development companies. It demonstrates the potential of integrating test agents to automate regression test cases and using explorative agents to test the robustness of game scenarios. However, the connection and interaction with XR-based games remains a complex area of test automation, requiring further research to resolve additional challenges related to 3D interaction and navigation.

Lack of technical integration solutions

A significant challenge in the field of game test automation is the absence of widely adopted solutions that support seamless connection and interaction with a broad range of XR game systems. While some research efforts, such as the VRTest framework [289], aim to provide extensible solutions, most game test automation still relies on unit tests integrated into game development frameworks [248]. Other solutions lack the necessary documentation to help integration with other game systems [308]. Moreover, methods based on pixel and image recognition may require multiple training for test agents to learn how to play scenarios of different games [158].

To overcome these limitations, this thesis utilizes the iv4XR framework. The iv4XR framework provides a set of interfaces that can be implemented as game plugins. These plugins can be integrated into TESTAR or other test automation tools, allowing a test agent to connect, obtain information, and interact with game objects within virtual environments.

However, given the technical diversity of game systems, developing these plugins for different games requires considerable technical effort, which harms the extrapolation of results.

A promising direction for improving iv4XR-based test automation is the integration of this framework with popular game engines such as Godot, Unity, or Unreal Engine. Although custom game objects vary significantly in their characteristics, many objects created in game engines share common properties and structures. For example, in the Godot game engine, Node3D⁵ objects are the foundation for 3D game objects, all of which possess similar position and rotation properties. Therefore, establishing a technical integration with Godot's internal system could at least enable access to positional data for game objects within Godot-based game environments.

Navigation in 3D game environments

Interaction with GUI systems typically involves mouse click and keyboard key input events and movements on a 2D screen. However, interaction in XR environments, such as 3D games, requires the interpretation and navigation of three-dimensional positions and orientations of all objects within the game state. This is essential to determine viable paths that a test agent can follow to reach a target position to explore or interact with game entities, while also identifying and avoiding potential obstructive obstacles.

As presented in this thesis, some game engines, such as LabRecruits built in Unity, can automatically generate a Navigation Mesh (NavMesh) based on the virtual geometry of objects in the game. This NavMesh provides walkable areas for agents by mapping surfaces. However, NavMeshes are primarily designed for navigation across 2D or surface-bound 3D environments⁶. This can lead to challenges in 3D games where empty space exists between disconnected surfaces, such as in open-world or space-based scenarios, where test agents may need to navigate through free space rather than along continuous surface terrain.

Another challenge arises when game environments do not support automatic NavMesh generation. In our early studies with Space Engineers, the game lacked any built-in NavMesh support, requiring us to use the geometry information of all surround observed entities to construct a navigation graph on the fly. While this solution enabled some level of pathfinding for scripted agents to position objectives, it introduced several limitations for a scriptless approach. Calculating explorable positions in runtime, along with handling dynamic orientation changes of test agents, led to significant computation costs and reduced navigation robustness in larger scenarios.

⁵https://docs.godotengine.org/en/stable/classes/class_node3d.html

⁶<https://docs.unity3d.com/Packages/com.unity.ai.navigation@1.1/manual/NavMeshSurface.html>

To address this, the Space Engineers team integrated a dedicated navigation layer that uses the positions of block entities forming grid structures. This allowed for the calculation of navigable positions on surfaces, enabling the TESTAR scriptless agent to efficiently navigate and interact with in-game entities using the A* algorithm from iv4XR. Despite these improvements, further research is needed to enhance the navigation layer to handle blocks larger than 1x1, which requires calculating their size and orientation to properly calculate obstructed spaces. Additionally, like in Unity-based environments, there is a need to research solutions for navigating between floating objects in space, such as spaceships flying to asteroids or remote space stations.

9.3 Abstraction strategies and state model inference

Abstraction strategies for state and action identification and state model inference processes are foundational pillars of this thesis. The effectiveness, efficiency, and efficacy of exploratory algorithms, distributed approaches, and delta GUI change detection solutions rely heavily on using abstraction strategies tailored to the specific complexities of the SUT.

This thesis contributes to the GUI test automation field by detailing how various abstraction strategies are implemented in TESTAR to achieve diverse testing purposes. We address the challenges of dynamism and non-determinism that impact scriptless exploration and model inference, proposing the extension of the main abstraction mechanisms with sub-strategies for identifying abstract states and actions. The objective is to mitigate these challenges by applying human expertise and SUT-specific knowledge.

A human tester must analyze the complexities of the SUT, identify widgets that contribute to dynamism and non-determinism, and configure an adequate abstraction strategy. This configuration includes the programmatic implementation of the sub-strategies and even configuring which actions to derive based on the objectives of the testing exploration or the inferred model.

Despite these efforts, certain challenges, such as non-determinism, may persist in some SUTs due to limited information on the GUI state. Addressing such non-determinism may require connecting to the SUT's internals, such as databases, or developing abstraction strategies that track predecessor state-action transitions across different states. In such cases, if the testing purpose does not require a deterministic model, exploratory algorithms may be extended to detect and deal with non-deterministic transitions.

While these solutions demand significant effort and may not be generalizable, they have proven effective and efficient during model inference. Moreover, they contribute to generating applicable models for further objectives like delta GUI change detection.

State model inference for 3D games

In this thesis, we applied an interactive explorer Action Selection Mechanism ASM to improve spatial coverage with two 3D games. This ASM uses the internal object identifiers and navigable positions provided by the game to memorize and prioritize unexplored actions. However, this non-persistent memory requires further research to integrate into the TESTAR state model.

XR game systems, particularly sandbox games like Space Engineers, present unique challenges due to the dynamic creation and destruction of game entities. Additionally, test agents constantly alter their positions while exploring 3D areas, which changes the observation range and leads to the constant observation of new entities and positions. These inherent complexities necessitate novel perspectives on abstraction, potentially extending the state model with a navigable layer to track explored areas and discovered entities [215]. However, further research is needed to explore the impact of open-world scenarios, such as those in Space Engineers, on the size and complexity of a navigable model layer.

9.4 Distributed State Model inference

The distributed state model inference technique improves the speed of model inference in automated scriptless GUI testing. The Distributed Model on a Shared SUT (DMSS) architecture allows multiple instances to infer a shared state model. The proposed shared knowledge algorithm allows TESTAR distributed instances to coordinate and improve the inference speed by dividing the unvisited actions to be explored.

Furthermore, this thesis also details challenges related to abstraction, dynamism, and non-determinism within the distributed inference process. Abstraction sub-strategies and algorithmic solutions are proposed to mitigate these challenges and reduce their impact on distributed model inference.

The contributions of this research, demonstrated through empirical experiments on two different open-source web SUTs, provide value to both academic researchers and industry practitioners. For the academic community, the findings offer insights into how a centralized model combined with a shared knowledge algorithm can enable the coordination and improve the efficiency of multiple instances in GUI test automation. This concept may serve for further research on optimizing automated testing processes. Additionally, the abstraction challenges and provided solutions may help future research efforts dealing with similar challenges.

For industrial companies, this distributed study can be used as inspiration for implementing distributed explorative test automation approaches for their applications. The study also

details the resources required for setting up a distributed architecture. Companies can then use this information to decide if increasing resources to improve the inference testing speed is beneficial.

Despite these advancements, further research is necessary to address the open directions and challenges presented below.

Concurrency in DMSS architectures

In DMSS architectures, multiple concurrent instances create, modify, or delete shared data in a common SUT. This can aggravate non-deterministic behavior during model inference. For instance, if one instance modifies a shared resource (such as a bank account balance or user information) while another instance simultaneously attempts to access or utilize that same data (for example, to initiate a bank transaction or submit a personal form), the resulting execution on the application server may yield unexpected or inconsistent outcomes, compared to tests performed with a single instance. However, this behavior can be employed to uncover potential concurrency errors within the SUT.

Missing SUT data in DMIS architectures

In DMIS (Distributed Model on Independent SUTs) architectures, each instance interacts with its own separate SUT, meaning there is no SUT data sharing. Although model knowledge is shared across instances, the independent data of each SUT can lead to inconsistencies when model information is used to follow a path of actions. For example, if Instance 1 creates a new bank account, this account exists only within the local SUT environment of Instance 1, and the creation of the account is not visible in the SUT of Instance 2. In such cases, it may be necessary for Instance 2 to reproduce a set of similar sequential actions to create the new bank account data in its local SUT environment. This may be a particularly relevant challenge in scenarios where distributed exploration is applied to SUTs that do not support DMSS architectures, which may be more common for desktop applications.

Traceability of models

In distributed testing architectures, traceability of test sequences generated in state models faces additional challenges due to concurrency in DMSS architectures and missing SUT data in DMIS architectures.

In the distributed TESTAR approach, multiple testing instances share model abstract information about the SUT, revealing which states and actions other instances are exploring. Nevertheless, each instance generates independent paths of test sequences. These paths are

essential for tracing the sequential path of actions from the initial loading of the SUT to a potential software failure state. While TESTAR might detect a failure in a state of the SUT, the process of tracing and reproducing this behavior can be inconsistent.

To address this traceability challenge, one possible solution is to extend the shared model knowledge to infer a centralized test sequence trace that integrates information from distributed sequences. For instance, using timestamps to order actions can help reconstruct the step-by-step test sequences and provide a traceable model.

9.5 Online oracles

This thesis has explored various types of oracles, demonstrating their practical benefits for fault detection in various industrial systems [237, 5, 40, 128, 43]. Collaborations with industrial partners have shown that investing effort into configuring test oracles within scriptless testing environments can reveal diverse types of unexpected software failures, thus improving the overall robustness of their SUTs. However, the research of invariant oracles, as well as action mechanisms combined with security oracles, remain areas that require further learning and refinement for integration into the TESTAR tool. Additionally, I have come to understand the significant complexity involved in defining test oracles for detecting faults in industrial 3D sandbox games, highlighting the need for further test automation research and techniques.

GUI online test oracles

In this thesis, implicit oracles have been extended to detect suspicious messages beyond the GUI data. These oracles have followed previous TESTAR studies to be fully integrated to monitor logs generated by the SUT and have been extended to identify suspicious messages in both the buffer outputs of desktop applications and the console of web browsers. These types of oracles are the easiest to configure for TESTAR users, yet they remain highly effective in detecting software failures, especially in GUI systems.

Through collaboration with I.D.B. Telematica B.V., we gained valuable insights into different types of bugs affecting web applications. This has enabled the preliminary analysis of these bugs and has provided a foundation for identifying which types of oracles could be integrated into TESTAR. For example, invariant and visual failure oracles can be integrated as general web oracles, allowing users to enable or disable them according to their specific needs. On the other hand, oracles designed to detect misspelling issues present a challenge, as their effectiveness and false positive rates depend on the number of languages supported by the SUT.

Additionally, research has been conducted into oracles aimed at identifying security faults. For instance, oracles that scan web headers to verify compliance with security policies function as monitoring oracles. Security oracles that detect XSS or SQL vulnerabilities mostly rely on regular expressions applied through implicit oracles to identify suspicious messages. In practice, the key challenge lies in deriving and selecting the appropriate actions that trigger payload injections to exploit potential security vulnerabilities.

Game online test oracles

Integrating TESTAR as a testing agent for the 3D industrial game Space Engineers provided both an exciting opportunity and a significant challenge. While working with a real-world game was motivating, the complexity of automating testing processes in such an environment also became apparent. After achieving robust navigability across large-scale game scenarios, a collaboration was established to analyze a vast number of bugs detected by game testers and reported by users of Space Engineers over several years.

These bug reports addressed various aspects of game testing. Many bugs were related to the precise construction of structures using specific block types, which required exact and sequential execution of actions. Additionally, some bugs only appeared in specific game scenarios, such as when a spatial door mechanism is built in space and its connected power source is abruptly destroyed, resulting in the door's malfunction. Trying to automate these tasks proved challenging and fell outside the primary objectives of scriptless testing.

Several game-related oracles require human intervention, particularly in validating graphical or audio elements. Testing multimedia components such as visual textures, user interface text displays, pixel particles, animations, lighting, and sounds is as essential as ensuring the game's blocks and items function correctly. However, verifying these visual and audio elements remains challenging for automated oracles. This validation may require a tridimensional perspective of all game objects and continuous observation of game objects' movements, often involving subjective human judgment.

At the end of this process, we decided to implement test oracles that check the basis invariants of game features. Nevertheless, there is a long way to research.

9.6 Delta GUI change detection oracle

In today's software projects, development methodologies emphasize rapid, iterative updates with frequent delta increments. As a result, it becomes imperative to integrate testing techniques that can automatically check the correctness of software changes to maintain

software quality. These techniques are essential to support developers and testers by providing them with analytics and visual results.

While code review has emerged and been embraced as a key strategy to ensure readability, consistency, and correctness in software projects, further research is needed into GUI change detection practices. This thesis conducts a systematic mapping of the literature, revealing that despite the existence of state-of-the-art delta GUI change detection techniques, their ideas and implementations remain scattered depending on the type of system or are not used for the main purpose of detecting and highlighting changes. This systematic mapping of the literature results contributes valuable insights into the current state of delta GUI change detection practices.

To tackle this limitation, this thesis presents a new delta GUI change detection approach using inferred models. By combining automated inferred GUI models with change detection algorithms for model transitions, the approach provides an interactive model that automatically highlights delta GUI changes. This interactive model can be useful for developers, testers, and other project contributors to detect added, removed, and changed functionalities.

The empirical evaluation demonstrates the versatility of the state model inference technique, showcasing its applicability to a diverse range of systems like desktop, web, and mobile applications. Notable examples include the detected Calibre web state that contains a typo issue that was already reported and fixed by their community and the successful detection of a Java object display issue reported by our team to the MyExpenses mobile software project.

The significance of these contributions can motivate academic and industrial practitioners to integrate this technique into their software projects or encourage them to enhance this innovative technique with further research. However, analyzing real and complex open-source systems reveals the potential benefits but also the challenges of employing inferred models for delta GUI change detection.

Determining GUI changes, of course, involves abstraction

The efficacy of the ChangeDetection tool, like any technique relying on model inference, depends on the design of adequate inference and abstraction strategies. The complexity of GUI systems, characterized by a multitude of states with dynamic behaviors, poses significant challenges that require extensive research and knowledge in inference and abstraction strategies when inferring state models. An inadequate inference or abstraction strategy can lead to erroneous change detection due to a lack of inferred GUI transitions if the model is not partially complete or due to dynamic alterations and nondeterministic behaviors. Implementing a Domain Specific Language (DSL) or a similar solution that allows the configurable level

of abstraction and inference strategies could help to deal with the intrinsic state explosion, dynamism, and non-determinism challenges of complex GUI systems that will remain.

GUI change detection noise

The existence of *noise* during the model traverse is a challenge that arises when comparing two models to detect changes. Introducing the capability for users to configure and utilize action descriptions during model comparison has proven to enhance the traversal of models by facilitating the comparison of subsequent transitions. Hence, in further implementations, integrating configuration functions or DSL instructions that allow users to ignore specific widgets (e.g., the static widgets from the OBS panel or the Calibre menu) could mitigate the propagation of change detection *noise* across the states of the merged model.

Empirical GUI change detection evaluation of large projects

When it comes to empirically evaluating the accuracy and inaccuracy of the delta GUI change detection technique, it becomes crucial to extend the assessment by incorporating metrics for false and true positives and negatives, accompanied by statistical analysis. However, conducting such evaluations demands a substantial understanding of the SUT to correlate the GUI with the underlying classes and code. Moreover, in certain systems like OBS desktop and Calibre web, where there are numerous commits and code changes between release versions, the manual inspection becomes impractical due to the sheer volume of changes. For this reason, the empirical evaluation review for these systems should involve a developer or tester expert in the SUT, or this evaluation should be performed on smaller SUT delta commit increments.

Pilot evaluation with students

Under the ENACTEST project [171], we conducted a preliminary evaluation of our change detection approach with six master's students. The experiments simulated a pull request with several delta commits of a web SUT. The students were tasked to determine if the commit code changes resulted in corresponding GUI changes and whether the changes indicated by the tool were correct or if the commits introduced GUI failures.

The feedback indicated that the change detection tool can help newcomers to a project visualize and understand the relationship between code and GUI during pull requests. Additionally, this evaluation results also pointed to technical aspects for visual improvements, like the need to highlight changes at the (Document Object Model) DOM level, even if the GUI visuals remain the same. For example, by extending the visualization of the changed

state with a textual description that summarizes detected changes and integrating Artificial Intelligence (AI) techniques such as Natural Language (NL) comments generation [208] adapted to DOM properties.

9.7 Academia-industry collaboration

During this thesis, we had the positive opportunity to collaborate on several European research projects with a diverse range of industrial companies involved in the development of complex desktop, web, mobile, and XR systems. These collaborations are fundamental for facilitating knowledge transfer and driving testing innovation. Furthermore, they offer mutual benefits. Academic obtain valuable insights into the companies internal testing environments and practical challenges. Moreover, academics have the opportunity to refine and validate their research solutions in real-world contexts. From the industry's perspective, these collaborations allow companies to assess the strengths and weaknesses of their quality testing processes. Additionally, they gain access to innovative solutions in software testing, which can complement and enhance their internal quality testing processes.

From an academic perspective, the TESTAR tool has particularly benefited from these collaborations. By applying this tool in real-world scenarios—ranging from embedded desktop systems and web-based port operations to mobile banking, enterprise systems, and 3D sandbox games—, we have been able to improve the integration of interaction plugins, enhance exploratory testing techniques, refine test oracles, and to polish the output results. These practical applications have not only driven innovation in scriptless testing but also motivated us to improve the development and testing workflows of the TESTAR tool itself.

In the following sections, we discuss some challenges that both industrial and academic partners should consider when engaging in such collaborations, drawing from our own experiences.

Dynamic nature of research projects

One of the key challenges in long-term industry-academia collaborations is the dynamic nature of research projects. When involved in a multi-year collaboration, the industry is more likely to change team members by incorporating new personnel or reallocating existing members to projects deemed of higher priority. Likewise, academic teams are subject to shifts, as bachelor, master, and PhD students or researchers may graduate or move on, requiring new participants to familiarize themselves with the project.

Flexibility, adaptability, and empathy are crucial to deal with these inevitable changes. Proactively anticipating changes in team compositions and recognizing that industry and

academic members may face varied future workloads enables smoother transitions and enhances the resilience of the collaboration. Effective collaboration requires strategic foresight, thorough planning, and meticulous documentation to facilitate knowledge transfer. By doing so, new team members—whether from industry, academia, or even external open-source contributors—can seamlessly integrate into the project and contribute productively.

Competing priorities

In industry-academia collaborations, competing priorities are an inherent challenge. While providing practical solutions to real-world problems is a shared goal, the academic side is also driven by the need to disseminate research publications. Writing research papers is a critical aspect of our work. It is one of the Key Performance Indicators (KPIs) used by European commissions and research departments to assess and approve research projects. Additionally, PhD students, who are often involved in these collaborations, need to demonstrate their capacity for knowledge acquisition, production, and dissemination by publishing in prestigious conferences and journals, which demands substantial effort.

On the academic side, we invest significant time, resources, and effort in researching, designing, and implementing novel techniques into pilot tools that offer practical value. Similarly, industry partners are motivated to apply these research insights, dedicating their resources to implement practical solutions while managing their internal company demands. The collaboration often yields effective results for specific use cases, such as a new XR testing framework for 3D game environments. However, these initial successes typically require further research and development to become scalable and widely applicable solutions for broader system environments.

Despite these efforts, the reality of industry-academia collaborations is that funding has an end. Once a research project ends, academic and industry partners are often required to shift and attend to other projects, changing the competing priorities and leaving limited time for maintaining and extending the developed solutions. This shift in priorities often means that further work relies on personal initiative and the availability of highly motivated individuals, who must also seek new funding opportunities to sustain the projects.

9.8 Scriptless test automation is a complementary solution

Manual testing approaches remain indispensable and highly effective due to the irreplaceable tacit knowledge brought by human testers. However, the growing complexity of systems and the time constraints for testing large applications have made some repetitive testing

techniques tedious and time-consuming. To address these challenges, GUI test automation has been extensively researched and developed to support and complement human testers.

Scripted GUI testing approaches—such as Selenium scripts used by KuveytTurk [5] and E-Dynamics [40], or the mobile Espresso scripts employed by ING [128]—are essential for automating regression tests, reducing the manual effort required for repetitive testing tasks. Nevertheless, one of the key empirical insights gained from working with industrial partners during this thesis is that scriptless testing offers a necessary complementary solution.

Unlike automated scripted approaches that follow predefined instructions, scriptless testing techniques generate test sequences dynamically, either randomly or using semi-random exploratory algorithms. This approach has proven to be effective in uncovering unforeseen states and revealing unexpected software failures.

Scriptless exploration can be further enhanced by incorporating domain-specific knowledge of the SUT, enabling SUT-specific actions and test oracles to improve coverage and fault detection capabilities. Nonetheless, attempting to implement too much domain-specific knowledge onto a scriptless testing tool, maybe trying to handle even sequential scripted test cases, could significantly increase maintenance costs, undermining the key advantages of scriptless testing.

In practice, scriptless testing is a valuable paradigm shift approach that should be considered a necessary solution in the testing process of industrial companies. Rather than replacing manual or scripted testing, scriptless testing complements these methods and should be integrated alongside them to maximize testing effectiveness.

9.9 Artificial Intelligence

Artificial Intelligence (AI) technologies, such as Large Language Models (LLMs), are demonstrating advanced testing capabilities for diverse testing objectives, including functional, robustness, and security testing at both unit and system levels [286, 160]. Adopting AI for testing could transition TESTAR from a traditional monkey testing tool to a more sophisticated, AI-driven, scriptless testing tool.

AI-TESTAR software interaction

AI technologies, such as object recognition neural networks (e.g., YOLO), can address the challenge of platform and API-independent widget recognition methods. Additionally, combining LLMs with web scanners can enable intelligent text injections tailored to the scripting, libraries, or server technologies identified in the SUT.

In XR game systems, AI can serve as a state environment observer, providing valuable insights into the 3D navigable paths a test agent can follow to interact with in-game entities. Additionally, AI-powered code generation tools can be integrated into frameworks like iv4XR to streamline the extension of game interfaces and facilitate smoother connections with the game SUT.

AI-TESTAR abstraction strategies and state model inference

Integrating AI for state and action identification methods requires further consideration and research. It is essential to define the role of an AI agent in abstraction strategies—whether it recommends the optimal combination of widget properties, acts as an intermediate entity for clustering states, informs about dynamic and non-deterministic behaviors, or provides programmatic filtering solutions for dynamic widgets.

Integration options for AI agents may also vary. For example, an online agent could be invoked after each state-action step during the testing process, though this could be resource-intensive. Alternatively, an offline agent might use the information of inferred concrete state models to provide abstraction recommendations. Furthermore, the type of information sent to the AI (e.g., widget properties, screenshots, or others) would depend on the chosen scriptless abstraction method.

AI-TESTAR exploratory algorithms

Exploration effectiveness in TESTAR could benefit from integrating AI-driven algorithms. Reinforcement Learning (RL) is still an ongoing area of research for improving exploratory strategies. Moreover, LLMs are being studied to be integrated with SUT requirements or user stories to guide Action Selection Mechanisms (ASM), helping to follow sequential or semi-sequential probabilistic paths.

AI-TESTAR test oracles

AI techniques can also enhance TESTAR's test oracles by using SUT requirements. AI can assist users in configuring oracles or serve as an active oracle agent to detect suspicious messages, verify invariant conditions, perform visual validations, check for spelling errors, and detect security flaws. An online AI agent could dynamically extract information from GUI states or web browser interfaces at runtime, applying online test oracles based on system requirements. Alternatively, an offline AI agent could analyze data from inferred models to check for these requirements [73].

Chapter 10

Conclusions and future work

In this thesis, our primary goal was to investigate the improvement of QA testing processes for desktop, web, mobile, and eXtended Reality (XR) applications through scriptless testing using the TESTAR tool. TESTAR is an open-source scriptless testing tool that has already demonstrated its capabilities in test coverage and fault detection across various industrial desktop and web systems, as well as Internet of Things (IoT) devices. However, to further evolve TESTAR into a solution for improving QA testing in industrial environments and other practical software projects, it was necessary to extend TESTAR's capabilities to interact with a broader range of software systems, enhance the efficiency and effectiveness of its exploratory algorithms, and further integrate oracles for assessing quality characteristics or detecting software failures.

To address these limitations and achieve the primary goal, we have enhanced the scriptless capabilities of TESTAR with several contributions that not only benefit the TESTAR project but may also be valuable to other academic researchers and industry practitioners working in scriptless testing or related test automation fields.

10.1 Contributions

The interaction capabilities of TESTAR have been extended to support Java Swing desktop applications using the Java Access Bridge, dynamic web applications via Selenium WebDriver, and Android and iOS mobile environments through Appium WebDriver. Additionally, TESTAR can now connect and interact with one experimental and one industrial XR 3D game, using the Intelligent Verification/Validation for Extended Reality Based Systems (iv4XR) framework. Furthermore, TESTAR functionality to interact with Windows desktop applications has been improved to handle multi-process applications. The automated interac-

tion with these systems can be executed on Virtual Machines (VMs) or Docker containers, depending on the underlying technology.

Contribution: *TESTAR interaction capabilities have been extended to interact with desktop, web, mobile, and XR systems.*

For the TESTAR state and action identification mechanism, we have researched, evaluated, and disseminated how dynamism, non-determinism, and state explosion challenges impact the process of calculating stable abstract identifiers. To address these challenges, we proposed an abstraction strategy that allows users to analyze and configure combinations of a main abstraction mechanism with a series of abstraction sub-strategies. This approach enables the generation of stable abstract identifiers to achieve specific testing objectives with specific Systems Under Test (SUT).

Contribution: *The challenges of dynamism, non-determinism, and state explosion that occur in large and complex systems have been documented, and solutions have been proposed.*

Contribution: *TESTAR offers a configurable approach that enables users to adapt abstraction strategies to their specific SUT and testing needs.*

For the TESTAR exploration capabilities, we integrated and evaluated a state model inference process that enhances exploration effectiveness by configuring a suitable level of abstraction. Additionally, we extended this process with a distributed approach, allowing multiple TESTAR instances to coordinate to improve the efficiency of the inference process. By employing grammar-based Action Selection Rules (ASRs), TESTAR can mimic human-like strategies, demonstrating improved effectiveness and efficiency in form-filling applications. An interactive Action Selection Mechanism (ASM) also enables a TESTAR agent to enhance spatial coverage when exploring and (dis)covering elements within the two XR 3D games.

Contribution: *TESTAR state model inference process enhances exploration effectiveness.*

Contribution: *TESTAR distributed approach improves the efficiency of the state model inference process.*

Contribution: *Grammar-based ASRs enable TESTAR to mimic human strategies, improving effectiveness and efficiency in form-filling tasks.*

Contribution: *A TESTAR agent can use interactive ASMs to improve spatial coverage effectiveness.*

The regular expression mechanism of TESTAR has been extended to detect suspicious messages in system logs and web browser console messages. By combining the derivation of actions for injecting XSS and SQL characters with the incorporation of corresponding patterns into its regular expressions, TESTAR is now capable of identifying XSS and SQL injection vulnerabilities. Additionally, a diverse set of programmatic oracles has been researched and integrated into TESTAR, enabling it to detect GUI data display faults, misspelling issues, and visual failures. For game systems, these programmatic test oracles primarily focus on evaluating pre- and post-conditions of game entity properties, allowing for the testing of functional aspects of XR games.

Contribution: *The regular expressions mechanism of TESTAR has been extended to detect suspicious messages in system logs and web browser console messages, including XSS and SQL injection vulnerabilities.*

Contribution: *TESTAR has been extended with programmatic oracles for detecting GUI data display faults, misspelling issues, and visual failures.*

Contribution: *TESTAR can assess functional aspects of XR games by using programmatic test oracles to evaluate pre- and post-conditions of game entity properties.*

Recognizing the importance of integrating delta GUI change detection techniques into software repositories to enhance software quality assurance, we have developed a novel open-source tool that uses TESTAR inferred state models. This tool enables an offline GUI change detection approach that detects and highlights GUI changes across state models from desktop, web, and mobile software systems.

Contribution: *A novel offline delta change detection solution uses the inferred GUI models from TESTAR to automatically detect and offer visualization of GUI changes.*

The insights gained for extending the TESTAR interaction, exploration, and oracle capabilities were achieved and evaluated through collaborations between academia and industry in real-world industrial environments. Ponsse and Prodevelop contributed significantly to advancements in connecting with multiprocess desktop GUI systems, enhancing TESTAR results reports, and evaluating the tool in CI environments. Collaborations with KuvveytTürk and ING motivated the integration of web and mobile GUI plugins into TESTAR. Keen Software House and GoodAI companies provided the opportunity to research the capabilities of TESTAR with an industrial 3D sandbox game. Additionally, I.D.B. Telematica B.V. played a crucial role in exploring the implementation of oracles for detecting GUI data display faults, misspelling issues, visual failures, and other oracle-related ideas. These collaborations demonstrated the TESTAR complementary capabilities for identifying unexpected errors not covered by traditional manual and scripted testing approaches, and increasing the test coverage of underlying SUT code.

Contribution: *TESTAR interaction, exploration, and oracle capabilities have been evaluated in real-world industrial environments.*

Contribution: *TESTAR has demonstrated to be a complementary testing solution compared to manual and scripted approaches in real-world industrial environments.*

In collaboration with Keen Software House and GoodAI companies, we extended this thesis beyond the TESTAR scriptless tool solution to research the application of a Behavior-Driven Development (BDD) approach for automating regression test cases. A BDD *agent* capable of following natural language instructions enabled the Space Engineers team to automate 236 tests from a total of 1322 regression tests that were previously executed manually.

Contribution: *A BDD agent can follow natural language instructions to automate regression test cases in an industrial 3D game.*

10.2 Publications

The research realized during this thesis has been presented and evaluated in various peer-reviewed national and international workshops, conferences, and journals from diverse software engineering areas, including information science, software systems, software testing, software verification and validation, software evaluation and assessment, software

reliability and security, software test automation, software quality, novel approaches to software engineering, and computer standards & interfaces. The research conducted for this thesis resulted in 20 publications, comprising 3 international journals, 11 international conferences, 3 international workshops, and 3 national conferences.

International Journals:

Title	TESTAR – scriptless testing through graphical user interface [276]
Authors	Tanja EJ Vos, Pekka Aho, Fernando Pastor Ricós , Olivia Rodriguez-Valdes, Ad Mulders
Published	Journal of Software Testing, Verification and Reliability (STVR), 2021
Impact	Journal Impact Factor in 2021 is 1.750
Ranking	Computer Science, Software Engineering in 2021 is 72 out of 110 (Q3)

Title	Distributed state model inference for scriptless GUI testing [240]
Authors	Fernando Pastor Ricós , Arend Slomp, Beatriz Marín, Pekka Aho, Tanja EJ Vos
Published	Journal of Systems and Software (JSS), 2023
Impact	Journal Impact Factor in 2023 is 3.7
Ranking	Computer Science, Software Engineering in 2023 is 28 out of 131 (Q1)

Title	Delta GUI Change Detection using inferred models [214]
Authors	Fernando Pastor Ricós , Beatriz Marín, Tanja EJ Vos, Rick Neeft, Pekka Aho
Published	Journal of Computer Standards & Interfaces (CS&I), 2024
Impact	Journal Impact Factor in 2023 is 4.1
Ranking	Computer Science, Software Engineering in 2023 is 20 out of 131 (Q1)

International Conferences:

Title	Deploying TESTAR to enable remote testing in an industrial CI pipeline: a case-based evaluation [237]
Authors	Fernando Pastor Ricós , Pekka Aho, Tanja Vos, Ismael Torres Boigues, Ernesto Calás Blasco, Héctor Martínez Martínez
Published	9th International Symposium on Leveraging Applications of Formal Methods (ISoLA), 2020
Ranking	CORE2023 C

Title	DECODER-DEveloper COmpanion for Documented and annotatEd code Reference [98]
Authors	Miriam Gil Pascual, Fernando Pastor Ricós , Maria Victoria Torres Bosch, Tanja Ernestina Vos
Published	14th International Conference on Research Challenges in Information Science (RCIS), 2020
Ranking	CORE2023 B

Title	IV4XR-Intelligent Verification/Validation for Extended Reality Based Systems [227]
Authors	ISWB Prasetya, Rui Prada, Tanja EJ Vos, Fitsum Kifetew, Frank Dignum, Jason Lander, Jean-yves Donnart, Alexandre Kazmierowski, Joseph Davidson, Fernando Pastor Ricós
Published	14th International Conference on Research Challenges in Information Science (RCIS), 2020
Ranking	CORE2023 B

Title	State model inference through the GUI using run-time test generation [189]
Authors	Ad Mulders, Olivia Rodriguez Valdes, Fernando Pastor Ricós , Pekka Aho, Beatriz Marín, Tanja EJ Vos
Published	16th International Conference on Research Challenges in Information Science (RCIS), 2022
Ranking	CORE2023 B

Title	Scriptless testing for extended reality systems [215]
Authors	Fernando Pastor Ricós
Published	16th International Conference on Research Challenges in Information Science (RCIS), 2022
Ranking	CORE2023 B

Title	Scriptless GUI Testing on Mobile Applications [128]
Authors	Thorn Jansen, Fernando Pastor Ricós , Yaping Luo, Kevin Van Der Vlist, Robbert Van Dalen, Pekka Aho, Tanja EJ Vos
Published	22nd International Conference on Software Quality, Reliability and Security (QRS), 2022
Ranking	CORE2023 C

Title	Using GUI Change Detection for Delta Testing [239]
Authors	Fernando Pastor Ricós , Rick Neeft, Beatriz Marín, Tanja EJ Vos, Pekka Aho
Published	17th International Conference on Research Challenges in Information Science (RCIS), 2023
Ranking	CORE2023 B

Title	Distributed state model inference for scriptless GUI testing [240]
Authors	Fernando Pastor Ricós , Arend Slomp, Beatriz Marín, Pekka Aho, Tanja EJ Vos
Published	Journal First article presented in the 27th International Conference on Evaluation and Assessment in Software Engineering (EASE), 2023
Ranking	CORE2023 A

Title	Scriptless Testing for an Industrial 3D Sandbox Game [238]
Authors	Fernando Pastor Ricós , Beatriz Marín, Tanja Vos, Joseph Davidson, Karel Hovorka
Published	19th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE), 2024
Ranking	CORE2023 B
Mention	High-quality paper invited for post-publication in Springer, ENASE 2024

Title	Grammar-based action selection rules for scriptless testing [122]
Authors	Lianne V Hufkens, Fernando Pastor Ricós , Beatriz Marin, Tanja EJ Vos
Published	5th International Conference on Automation of Software Test (AST), 2024
Ranking	CORE2023 C

Title	An Industrial Experience Leveraging the iv4XR Framework for BDD Testing of a 3D Sandbox Game [216]
Authors	Fernando Pastor Ricós , Beatriz Marín, ISWB Prasetya, Tanja EJ Vos, Joseph Davidson, Karel Hovorka
Published	18th International Conference on Research Challenges in Information Science (RCIS), 2024
Ranking	CORE2023 B
Award	Best Paper Award, RCIS 2024

International Workshops:

Title	An agent-based approach to automated game testing: an experience report [226]
Authors	ISWB Prasetya, Fernando Pastor Ricós , Fitsum Meshesha Kifetew, Davide Prandi, Samira Shirzadehhajimahmood, Tanja EJ Vos, Premysl Paska, Karel Hovorka, Raihana Ferdous, Angelo Susi, Joseph Davidson
Published	13th International Workshop on Automating Test Case Design, Selection and Evaluation (A-TEST), 2022
Co-located	European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2022
Ranking	CORE2023 A*

Title	Exploring students' opinion on software testing courses [50]
Authors	Felix Cammaerts, Porfirio Tramontana, Ana CR Paiva, Nuno Flores, Fernando Pastor Ricós , Monique Snoeck
Published	2nd International Workshop on evaluation and assessment in software Engineers' Education and Training (LEARNER), 2024
Co-located	28th International Conference on Evaluation and Assessment in Software Engineering (EASE), 2024
Ranking	CORE2023 A

Title	Bridging the gap between industry and academia and raising awareness of software testing bias through ENACTEST [213]
Authors	Fernando Pastor Ricós , Niels Doorn, Beatriz Marín, Tanja EJ Vos
Published	11th ACM Celebration of Women in Computing: womENCourage, 2024

National Conferences:

Title	Continuous piloting of an open source test automation tool in an industrial environment [11]
Authors	Pekka Aho, Tanja EJ Vos, Sami Ahonen, Tomi Piirainen, Perttu Moilanen, Fernando Pastor Ricós
Published	XXIV Jornadas de Ingeniería del Software y Bases de Datos (JISBD), 2019

Title	Applying scriptless test automation on web applications from the financial sector [5]
Authors	Pekka Aho, Govert Buijs, Abdurrahman Akin, Serafettin Senturk, Fernando Pastor Ricós , Stijn de Gouw, Tanja Ernestina Vos
Published	XXV Jornadas de Ingeniería del Software y Bases de Datos (JISBD), 2021

Title	Evaluating TESTAR's effectiveness through code coverage [274]
Authors	Aaron van der Brugge, Fernando Pastor Ricós , Pekka Aho, Beatriz Marín, Tanja Ernestina Vos
Published	XXV Jornadas de Ingeniería del Software y Bases de Datos (JISBD), 2021

10.3 Future work

This thesis has made valuable contributions to the TESTAR tool and, in general, to the scriptless software testing field. However, these thesis results do not represent a closed work. There remain several conceptual and technical directions that still require further research to continue advancing in the scriptless testing and test automation areas. The discussion section mentioned further research directions and challenges to continue the present thesis work. The following list summarizes a series of activities that are currently under research and work or that we are considering studying in the near future:

- Use object and optical recognition tools [299, 234] to overcome the technical limitations that test automation tools face when existing automation APIs or frameworks are either lacking or insufficient. The TESTAR project has two research directions using the Ultralytics YOLO tool for object recognition [132] and the Tesseract engine for optical character recognition [178]. These AI technologies can be combined to address challenges associated with scriptless testing Linux GUI desktop systems.
- Investigate the integration of the iv4XR framework with game engines such as Unity, Godot, or Unreal Engine to evaluate the automation capabilities of test agents. The iv4XR framework provides flexible interfaces that allow game developers to create custom iv4XR plugins tailored to the internal technical needs of their games. However, this requires considerable technical effort. To streamline this process, future work could focus on developing an iv4XR engine framework that can automatically detect the properties of game entities.
- Research the role of AI agents in supporting test automation tools to use abstraction mechanisms for identifying GUI and XR states and actions. An AI agent could play an active role during runtime exploration, assisting in clustering states when detecting dynamic or non-deterministic behavior. Alternatively, it could function offline, analyzing inferred models and recommending suitable combinations of properties and sub-strategies for specific SUTs and testing needs.
- Extend the state model with an additional layer that enables TESTAR to track navigable positions and reachable entities in-game open spaces. This extension could allow a TESTAR *agent* to remember which positions and entities, beyond its current observation state, are reachable. It could also facilitate the exploration of game environments that lack built-in navigation meshes.
- Implement a Domain-Specific Language (DSL) or a similar solution to enhance the usability of TESTAR in terms of configuration, readability, and maintainability. A DSL can provide users with a more human-readable way to specify the derivation of actions, prepare ASM decisions, and configure oracles. Moreover, it may offer a flexible manner for configuring a suitable abstraction strategy in TESTAR or be integrated into the change detection tool for selecting which widgets to ignore, thus reducing the change detection *noise*. Finally, incorporating AI agents, such as LLMs, could leverage the DSL to enhance the scriptless exploration capabilities of TESTAR for diverse decision-making processes.

- Study the sustainability impact of industrial software systems and the software testing processes used to assess the quality of these systems. Nowadays, software development and testing processes often lack mechanisms and awareness aimed at evaluating the environmental and social impact of their products. Our plan includes collaborating with partners in autonomous driving and port management through European research projects to assess the sustainability characteristics of teleoperated vehicles and port water quality software.
- Study the effectiveness and efficiency of TESTAR's scriptless capabilities compared to other state-of-the-art tools and solutions. Throughout this thesis, TESTAR has demonstrated its value as a complementary solution to manual and scripted testing methods. Building on these findings, we plan to continue our comparative studies to include model-based testing tools and other scriptless solutions. These future findings will help to showcase TESTAR's strengths and the features that require improvement.
- Integrate AI agents into TESTAR to enable the tool to cover SUT user-stories. Nowadays, automating user-stories demands significant manual effort to create and maintain sequential scripts. While scriptless testing can be used to automate the assessment of quality SUT characteristics like reliability and security, it would require too much domain-specific knowledge and effort to cover user-stories. However, with recent advancements in AI, intelligent agents can be integrated into TESTAR to overcome this limitation. By combining the advantages of scriptless automation with AI-driven decision-making, these agents can guide TESTAR in efficiently covering user-stories, shifting the testing paradigm from test scripts development to AI-driven testing solutions.

References

- [1] Adachi, Y., Tanno, H., and Yoshimura, Y. (2020). A method to mask dynamic content areas based on positional relationship of screen elements for visual regression testing. In *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 1755–1760. IEEE.
- [2] Adamo, D., Nurmuradov, D., Piparia, S., and Bryce, R. (2018a). Combinatorial-based event sequence testing of android applications. *IST journal*, 99:98–117.
- [3] Adamo, D., Nurmuradov, D., Piparia, S., and Bryce, R. (2018b). Combinatorial-based event sequence testing of android applications. *Information and Software Technology*, 99:98–117.
- [4] Agrawal, N., Ananthanarayanan, R., Gupta, R., Joshi, S., Krishnapuram, R., and Negi, S. (2004). The eshopmonitor: A comprehensive data extraction tool for monitoring web sites. *IBM journal of research and development*, 48(5.6):679–692.
- [5] Aho, P., Buijs, G., Akin, A., Senturk, S., Pastor-Ricós, F., de Gouw, S., and Vos, T. E. (2021). Applying scriptless test automation on web applications from the financial sector. *Actas de las XXV Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2021)*, pages 1–4.
- [6] Aho, P., Menz, N., Rätty, T., and Schieferdecker, I. (2011). Automated java gui modeling for model-based testing purposes. In *2011 Eighth International Conference on Information Technology: New Generations*, pages 268–273. IEEE.
- [7] Aho, P., Suarez, M., Kanstrén, T., and Memon, A. M. (2013). Industrial adoption of automatically extracted gui models for testing. In *EESSMOD@ MoDELS*, pages 49–54.
- [8] Aho, P., Suarez, M., Kanstrén, T., and Memon, A. M. (2014). Murphy tools: Utilizing extracted gui models for industrial software testing. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, pages 343–348. IEEE.
- [9] Aho, P., Suarez, M., Memon, A., and Kanstrén, T. (2015). Making gui testing practical: Bridging the gaps. In *2015 12th International Conference on Information Technology-New Generations*, pages 439–444. IEEE.
- [10] Aho, P. and Vos, T. (2018). Challenges in automated testing through graphical user interface. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 118–121. IEEE.

- [11] Aho, P., Vos, T. E., Ahonen, S., Piirainen, T., Moilanen, P., and Ricós, F. P. (2019). Continuous piloting of an open source test automation tool in an industrial environment. In *Las Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, pages 1–4. Sistedes.
- [12] Alégroth, E., Karlsson, A., and Radway, A. (2018). Continuous integration and visual gui testing: Benefits and drawbacks in industrial practice. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 172–181. IEEE.
- [13] Almenar, F., Esparcia-Alcázar, A. I., Martínez, M., and Rueda, U. (2016). Automated testing of web applications with testar: Lessons learned testing the odoo tool. In *Search Based Software Engineering: 8th International Symposium, SSBSE 2016, Raleigh, NC, USA, October 8-10, 2016, Proceedings 8*, pages 218–223. Springer.
- [14] Amalfitano, D., Amatucci, N., Memon, A. M., Tramontana, P., and Fasolino, A. R. (2017). A general framework for comparing automatic testing techniques of android mobile apps. *Journal of Systems and Software*, 125:322–343.
- [15] Andreessen, M. et al. (2011). Why software is eating the world. *Wall Street Journal*, 20(2011):C2.
- [16] Andrews, K., Wohlfahrt, M., and Wurzinger, G. (2009). Visual graph comparison. In *13th International Conference Information Visualisation*, pages 62–67. IEEE.
- [17] Anjum, M. S. and Ryan, C. (2021). Seeding grammars in grammatical evolution to improve search-based software testing. *SN Computer Science*, 2(4):280.
- [18] Arcuri, A. and Briand, L. (2011). A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 1–10. IEEE.
- [19] Arcuri, A. and Briand, L. (2014). A hitchhiker’s guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250.
- [20] Arif, B. (2019). Grammar-based white-box testing via automated constraint path generation. In *IEMIS, Volume 1*, pages 65–77. Springer.
- [21] Armstrong, R. A. (2014). When to use the bonferroni correction. *Ophthalmic and Physiological Optics*, 34(5):502–508.
- [22] Aslan, Ö., Aktuğ, S. S., Ozkan-Okay, M., Yilmaz, A. A., and Akin, E. (2023). A comprehensive review of cyber security vulnerabilities, threats, attacks, and solutions. *Electronics*, 12(6):1333.
- [23] Azim, T. and Neamtiu, I. (2013). Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 641–660.
- [24] Bahaweres, R. B., Oktaviani, E., Wardhani, L. K., Hermadi, I., Suroso, A., Solihin, I. P., and Arkeman, Y. (2020). Behavior-driven development (bdd) cucumber katalon for automation gui testing case cura and swag labs. In *ICIMCIS*, pages 87–92.

- [25] Balaji, S. and Murugaiyan, M. S. (2012). Waterfall vs. v-model vs. agile: A comparative study on sdlc. *International Journal of Information Technology and Business Management*, 2(1):26–30.
- [26] Banerjee, I., Nguyen, B., Garousi, V., and Memon, A. (2013). Graphical user interface (gui) testing: Systematic mapping and repository. *Information and Software Technology*, 55(10):1679–1694.
- [27] Baresi, L. and Young, M. (2001). Test oracles, technical report cis-tr-01-02. *University of Oregon, Dept. of Computer and Information Science*.
- [28] Barr, E. T., Harman, M., McMinn, P., Shahbaz, M., and Yoo, S. (2014). The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, 41(5):507–525.
- [29] Barth, A., Caballero, J., and Song, D. (2009). Secure content sniffing for web browsers, or how to stop papers from reviewing themselves. In *2009 30th IEEE Symposium on Security and Privacy*, pages 360–371. IEEE.
- [30] Bates, D., Barth, A., and Jackson, C. (2010). Regular expressions considered harmful in client-side xss filters. In *Proceedings of the 19th international conference on World wide web*, pages 91–100.
- [31] Bauer, F. L. (2000). A computer pioneer’s talk: pioneering work in software during the 50s in central europe. In *History of Computing: Software Issues: International Conference on the History of Computing, ICHC 2000 April 5–7, 2000 Heinz Nixdorf MuseumsForum Paderborn, Germany*, pages 11–22. Springer.
- [32] Bauersfeld, S., de Rojas, A., and Vos, T. E. (2014a). Evaluating rogue user testing in industry: an experience report. In *2014 IEEE eighth international conference on research challenges in information science (RCIS)*, pages 1–10. IEEE.
- [33] Bauersfeld, S. and Vos, T. E. (2012a). Guitest: a java library for fully automated gui robustness testing. In *Proceedings of the 27th ieee/acm international conference on automated software engineering*, pages 330–333.
- [34] Bauersfeld, S. and Vos, T. E. (2012b). A reinforcement learning approach to automated gui robustness testing. In *4 th Symposium on Search Based-Software Engineering*, page 7.
- [35] Bauersfeld, S. and Vos, T. E. (2014). User interface level testing with testar: what about more sophisticated action specification and selection? In *7th Edition Advanced Techniques and Tools for Software Evolution*, pages 60–78. CEUR-WS. org.
- [36] Bauersfeld, S., Vos, T. E., Condori-Fernández, N., Bagnato, A., and Brosse, E. (2014b). Evaluating the testar tool in an industrial case study. In *Proceedings of the 8th acm/ieee international symposium on empirical software engineering and measurement*, pages 1–9.
- [37] Bessghaier, N., Soui, M., Kolski, C., and Chouchane, M. (2021). On the detection of structural aesthetic defects of android mobile user interfaces with a metrics-based tool. *ACM Transactions on Interactive Intelligent Systems (TiiS)*, 11(1):1–27.

- [38] Binamungu, L. P. and Maro, S. (2023). Behaviour driven development: A systematic mapping study. *Journal of Systems and Software*, page 111749.
- [39] Böhme, M. and Paul, S. (2015). A probabilistic analysis of the efficiency of automated software testing. *IEEE Transactions on Software Engineering*, 42(4):345–360.
- [40] Bons, A., Marín, B., Aho, P., and Vos, T. (2023). Scripted and scriptless gui testing for web applications: An industrial case. *IST journal*, 158:107172.
- [41] Borges, N., Hotzkow, J., and Zeller, A. (2018). Droidmate-2: a platform for android test generation. In *33rd ASE*, pages 916–919. IEEE.
- [42] Borque, P. and Fairley, R. (2014). Guide to the software engineering body of knowledge version 3.0. *IEEE Computer Society Staff*.
- [43] Bouwmeester, R. (2023). Research case study improve bug finding ability of testar by using historical bug database of digioffice. Master’s thesis, Open Universiteit.
- [44] Brach, P., Chrzaszcz, J., Jabłonowski, J., and Świątły, J. (2011). A distributed service oriented system for gui map generation. In *Proceedings of the 12th International Conference on Computer Systems and Technologies*, pages 69–74.
- [45] Bruns, A., Kornstadt, A., and Wichmann, D. (2009). Web application tests with selenium. *IEEE software*, 26(5):88–91.
- [46] Bryant, C., Yuan, Z., Qorib, M. R., Cao, H., Ng, H. T., and Briscoe, T. (2023). Grammatical error correction: A survey of the state of the art. *Computational Linguistics*, 49(3):643–701.
- [47] Budtmo (2024). Android in docker solution with novnc supported and video recording. Last accessed: 1 Sep 2024.
- [48] Bures, M. (2014). Change detection system for the maintenance of automated testing. In *Testing Software and Systems: 26th IFIP WG 6.1 International Conference, ICTSS 2014, Madrid, Spain, September 23-25, 2014. Proceedings 26*, pages 192–197. Springer.
- [49] Calibre (2006, 2023). Web app for browsing, reading and downloading ebooks. Last accessed: 9 Dec 2022.
- [50] Cammaerts, F., Tramontana, P., Paiva, A. C., Flores, N., Pastor Ricós, F., and Snoeck, M. (2024). Exploring students’ opinion on software testing courses. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, pages 570–579.
- [51] Canny, A., Palanque, P., and Navarre, D. (2020). Model-based testing of gui applications featuring dynamic instantiation of widgets. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 95–104. IEEE.
- [52] Cao, Y., Wu, G., Chen, W., and Wei, J. (2018). Crawlroid: Effective model-based gui testing of android apps. In *Proceedings of the 10th Asia-Pacific Symposium on Internetware*, pages 1–6.

- [53] Causevic, A., Sundmark, D., and Punnekkat, S. (2010). An industrial survey on contemporary aspects of software testing. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 393–401. IEEE.
- [54] Chahim, H., Duran, M., and Vos, T. (2018). Challenging testar in an industrial setting: the rail sector. *Information Systems Development: Designing Digitalization (ISD2018)*.
- [55] Chahim, H., Duran, M., Vos, T. E., Aho, P., and Condori Fernandez, N. (2020). Scriptless testing at the gui level in an industrial setting. In *International Conference on Research Challenges in Information Science*, pages 267–284. Springer.
- [56] Chang, N., Wang, L., Pei, Y., Mondal, S. K., and Li, X. (2018). Change-based test script maintenance for android apps. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 215–225. IEEE.
- [57] Charette, R. N. (2005). Why software fails [software failure]. *IEEE spectrum*, 42(9):42–49.
- [58] Chen, C., Cui, B., Ma, J., Wu, R., Guo, J., and Liu, W. (2018). A systematic review of fuzzing techniques. *Computers & Security*, 75:118–137.
- [59] Chen, L. and Avizienis, A. (1978). N-version programming: A fault-tolerance approach to reliability of software operation. In *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8)*, volume 1, pages 3–9.
- [60] Choudhary, S. R., Gorla, A., and Orso, A. (2015). Automated test input generation for android: Are we there yet?(e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 429–440. IEEE.
- [61] Choudhary, S. R., Zhao, D., Versee, H., and Orso, A. (2011). Water: Web application test repair. In *Proceedings of the First International Workshop on End-to-End Test Script Engineering*, pages 24–29.
- [62] Choudhury, N. (2014). World wide web and its journey from web 1.0 to web 4.0. *International Journal of Computer Science and Information Technologies*, 5(6):8096–8100.
- [63] Collins, E., Neto, A., Vincenzi, A., and Maldonado, J. (2021). Deep reinforcement learning based android application gui testing. In *Proceedings of the XXXV Brazilian Symposium on Software Engineering*, pages 186–194.
- [64] Cooper, K. M. (2021). *Software engineering perspectives in computer game development*. CRC Press.
- [65] Coppola, R. and Alégroth, E. (2022). A taxonomy of metrics for gui-based testing research: A systematic literature review. *IST journal*, page 107062.
- [66] Coppola, R. and Alégroth, E. (2022). A taxonomy of metrics for gui-based testing research: A systematic literature review. *Information and Software Technology*, 152:107062.
- [67] Coppola, R., Morisio, M., and Torchiano, M. (2018). Mobile gui testing fragility: a study on open-source android applications. *IEEE Transactions on Reliability*, 68(1):67–90.

- [68] Crespo-Martínez, I. S., Campazas-Vega, A., Guerrero-Higueras, Á. M., Riego-DelCastillo, V., Álvarez-Aparicio, C., and Fernández-Llamas, C. (2023). Sql injection attack detection in network flow data. *Computers & Security*, 127:103093.
- [69] CSTI (2013, 2022). Shopizer v2.17 java e-commerce software. Last accessed: 7 Apr 2022.
- [70] Cui, X. and Shi, H. (2011). A*-based pathfinding in modern computer games. *International Journal of Computer Science and Network Security*, 11(1):125–130.
- [71] Dalal, Z., Dash, S., Dave, P., Francisco-Revilla, L., Furuta, R., Karadkar, U., and Shipman, F. (2004). Managing distributed collections: evaluating web page changes, movement, and replacement. In *Proceedings of the 4th ACM/IEEE-CS joint conference on Digital libraries*, pages 160–168.
- [72] Dallmeier, V., Burger, M., Orth, T., and Zeller, A. (2012). Webmate: a tool for testing web 2.0 applications. In *Workshop on JavaScript Tools*, pages 11–15.
- [73] de Gier, F., Kager, D., de Gouw, S., and Vos, E. T. (2019). Offline oracles for accessibility evaluation with the testar tool. In *2019 13th International Conference on Research Challenges in Information Science (RCIS)*, pages 1–12. IEEE.
- [74] Dees, I., Wynne, M., and Hellesoy, A. (2013). *Cucumber Recipes: Automate Anything with BDD Tools and Techniques*. Pragmatic Bookshelf.
- [75] Deguchi, A., Hirai, C., Matsuoka, H., Nakano, T., Oshima, K., Tai, M., and Tani, S. (2020). *What Is Society 5.0?*, pages 1–23. Springer Singapore, Singapore.
- [76] Denning, P. J. (1992). What is software quality? *Communications of the ACM*, 35(1):13–15.
- [77] Dijkstra, E. W. (2022). On the reliability of programs. In *Edsger Wybe Dijkstra: His Life, Work, and Legacy*, pages 359–370. Association for Computing Machinery.
- [78] Eloff, J., Bihina Bella, M., Eloff, J., and Bella, M. B. (2018). Software failures: An overview. *Software Failure Investigation: A Near-Miss Analysis Approach*, pages 7–24.
- [79] Eskonen, J., Kahles, J., and Reijonen, J. (2020). Automating gui testing with image-based deep reinforcement learning. In *Int. Conf. on Autonomic Computing and Self-Organizing Systems*, pages 160–167. IEEE.
- [80] Esparcia-Alcazar, A., Almenar, F. F., Martinez, M., Rueda, U. U., and Vos, T. (2016). Q-learning strategies for action selection in the testar automated testing tool. In *6th International Conference on Metaheuristic and Nature inspired Computing*, pages 174–180.
- [81] Esparcia-Alcázar, A. I., Almenar, F., Rueda, U., and Vos, T. E. (2017). Evolving rules for action selection in automated testing via genetic programming—a first approach. In *European Conference on the Applications of Evolutionary Computation*, pages 82–95. Springer.

- [82] Esparcia-Alcázar, A. I., Almenar, F., Vos, T. E., and Rueda, U. (2018). Using genetic programming to evolve action selection rules in traversal-based automated software testing: results obtained with the testar tool. *Memetic Computing*, 10:257–265.
- [83] Ewusi-Mensah, K. (2003). *Software development failures*. Mit Press.
- [84] Felderer, M., Büchler, M., Johns, M., Brucker, A. D., Breu, R., and Pretschner, A. (2016). Security testing: A survey. In *Advances in Computers*, volume 101, pages 1–51. Elsevier.
- [85] Feldt, R. and Magazinius, A. (2010). Validity threats in empirical software engineering research—an initial survey. In *Seke*, pages 374–379.
- [86] Fernández-Bonilla, F., Gijón, C., and De la Vega, B. (2022). E-commerce in spain: Determining factors and the importance of the e-trust. *Telecommunications Policy*, 46(1):102280.
- [87] Fisher, J., Koning, D., and Ludwigsen, A. (2013). Utilizing atlassian jira for large-scale software development management. Technical report, LLNL, Livermore, CA (United States).
- [88] Flesca, S., Furfaro, F., and Masciari, E. (2001). Monitoring web information changes. In *Proceedings International Conference on Information Technology: Coding and Computing*, pages 421–425. IEEE.
- [89] for Standardization, I. O. (2023). Iso/iec 25010:2023, systems and software engineering — systems and software quality requirements and evaluation (square) — product quality model. ISO/IEC 25010:2023.
- [90] Francisco-Revilla, L., Shipman, F., Furuta, R., Karadkar, U., and Arora, A. (2001a). Managing change on the web. In *Proceedings of the 1st ACM/IEEE-CS joint conference on Digital libraries*, pages 67–76.
- [91] Francisco-Revilla, L., Shipman III, F. M., Furuta, R., Karadkar, U., and Arora, A. (2001b). Perception of content, structure, and presentation changes in web-based hypertext. In *Proceedings of the 12th ACM Conference on Hypertext and Hypermedia*, pages 205–214.
- [92] Fulcini, T., Coppola, R., Torchiano, M., and Ardito, L. (2023). An analysis of widget layout attributes to support android gui-based testing. In *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 117–125. IEEE.
- [93] Furia, C. A., Feldt, R., and Torkar, R. (2019). Bayesian data analysis in empirical software engineering research. *IEEE Transactions on Software Engineering*, 47(9):1786–1810.
- [94] Gao, Z., Chen, Z., Zou, Y., and Memon, A. M. (2015a). Sitar: Gui test script repair. *Ieee transactions on software engineering*, 42(2):170–186.

- [95] Gao, Z., Fang, C., and Memon, A. M. (2015b). Pushing the limits on automation in gui regression testing. In *2015 IEEE 26th international symposium on software reliability engineering (ISSRE)*, pages 565–575. IEEE.
- [96] Garcia, A. F., Rubira, C. M., Romanovsky, A., and Xu, J. (2001). A comparative study of exception handling mechanisms for building dependable object-oriented software. *Journal of systems and software*, 59(2):197–222.
- [97] Giachetti, G., de la Vara, J. L., and Marín, B. (2023). A model-driven approach to adopt good practices for agile process configuration and certification. *Computer Standards & Interfaces*, 86:103737.
- [98] Gil Pascual, M., Pastor Ricos, F., Torres Bosch, V., and Vos, T. E. (2020). Decoder - developer companion for documented and annotated code reference. In *14th International Conference on Research Challenges in Information Science (RCIS), Limassol, Cyprus, September 23-25, 2020*, volume 385, pages 643–644. Springer.
- [99] Gillies, A. (2011). *Software quality: theory and management*. Lulu. com.
- [100] Grechanik, M., Mao, C. W., Baisal, A., Rosenblum, D., and Hossain, B. M. (2018). Differencing graphical user interfaces. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 203–214. IEEE.
- [101] Grechanik, M., Xie, Q., and Fu, C. (2009). Maintaining and evolving gui-directed test scripts. In *2009 IEEE 31st International Conference on Software Engineering*, pages 408–418. IEEE.
- [102] Grigorescu, A., Pelinescu, E., Ion, A. E., and Dutcas, M. F. (2021). Human capital in digital economy: An empirical analysis of central and eastern european countries from the european union. *Sustainability*, 13(4):2020.
- [103] Gu, T., Cao, C., Liu, T., Sun, C., Deng, J., Ma, X., and Lü, J. (2017). Aimdroid: Activity-insulated multi-level automated testing for android applications. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 103–114. IEEE.
- [104] Guo, H.-F. and Qiu, Z. (2014). A dynamic stochastic model for automatic grammar-based test generation. *Software: Practice and Experience*, 45(11):1519–1547.
- [105] Gupta, S. and Gupta, B. B. (2017). Cross-site scripting (xss) attacks and defense mechanisms: classification and state-of-the-art. *International Journal of System Assurance Engineering and Management*, 8:512–530.
- [106] Gurung, G., Shah, R., and Jaiswal, D. P. (2020). Software development life cycle models-a comparative study. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, 6(4):30–37.
- [107] Hadlington, L. and Scase, M. O. (2018). End-user frustrations and failures in digital technology: exploring the role of fear of missing out, internet addiction and personality. *Heliyon*, 4(11).

- [108] Hallé, S., Bergeron, N., Guerin, F., and Le Breton, G. (2015). Testing web applications through layout constraints. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–8. IEEE.
- [109] Hamill, M. and Goseva-Popstojanova, K. (2009). Common trends in software fault and failure data. *IEEE Transactions on Software Engineering*, 35(4):484–496.
- [110] Hammoudi, M., Rothermel, G., and Stocco, A. (2016a). Waterfall: An incremental approach for repairing record-replay tests of web applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 751–762.
- [111] Hammoudi, M., Rothermel, G., and Tonella, P. (2016b). Why do record/replay tests of web applications break? In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 180–190. IEEE.
- [112] Hannousse, A., Yahiouche, S., and Nait-Hamoud, M. C. (2024). Twenty-two years since revealing cross-site scripting attacks: a systematic mapping and a comprehensive survey. *Computer Science Review*, 52:100634.
- [113] Hao, S., Liu, B., Nath, S., Halfond, W. G., and Govindan, R. (2014). Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 204–217.
- [114] Haverty, R. (2005). New accessibility model for microsoft windows and cross platform development. *ACM SIGACCESS Accessibility and Computing*, 1(82):11–17.
- [115] Hemmati, H. (2015). How effective are code coverage criteria? In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 151–156. IEEE.
- [116] Herkert, J., Borenstein, J., and Miller, K. (2020). The boeing 737 max: Lessons for engineering ethics. *Science and engineering ethics*, 26:2957–2974.
- [117] Hoda, R., Salleh, N., and Grundy, J. (2018). The rise and evolution of agile software development. *IEEE software*, 35(5):58–63.
- [118] Hoebert, J. (2022). Using gui testing to automate website security analysis. Master’s thesis, Open Universiteit.
- [119] Hofer, B., Peischl, B., and Wotawa, F. (2009). Gui savvy end-to-end testing with smart monkeys. In *2009 ICSE Workshop on Automation of Software Test*, pages 130–137. IEEE.
- [120] Honest, N. (2019). Role of testing in software development life cycle. *International Journal of Computer Sciences and Engineering*, 7(5):886–889.
- [121] Hooda, I. and Chhillar, R. S. (2015). Software test process, testing types and techniques. *International Journal of Computer Applications*, 111(13).
- [122] Hufkens, L. V., Pastor Ricos, F., Marín, B., and Vos, T. E. (2024). Grammar-based action selection rules for scriptless testing. In *Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024)*, pages 56–65.

- [123] Itkonen, J., Mantyla, M. V., and Lassenius, C. (2009). How do testers do it? an exploratory study on manual testing practices. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 494–497. IEEE.
- [124] Itkonen, J., Mäntylä, M. V., and Lassenius, C. (2012). The role of the tester’s knowledge in exploratory software testing. *IEEE Transactions on Software Engineering*, 39(5):707–724.
- [125] Ivanković, M., Petrović, G., Just, R., and Fraser, G. (2019). Code coverage at google. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 955–963.
- [126] Jacob, J., Sachde, A., and Chakravarthy, S. (2005). Cx-diff: a change detection algorithm for xml content and change visualization for webvigil. *Data & Knowledge Engineering*, 52(2):209–230.
- [127] JaCoCo (2006, 2022). Java code coverage. Last accessed: 7 Apr 2022.
- [128] Jansen, T., Ricós, F. P., Luo, Y., Van Der Vlist, K., Van Dalen, R., Aho, P., and Vos, T. E. (2022). Scriptless gui testing on mobile applications. In *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*, pages 1103–1112. IEEE.
- [129] Jatowt, A., Kawai, Y., and Tanaka, K. (2009). Browsing assistant for changing pages. In *Intelligent Agents in the Evolution of Web and Applications*, pages 137–160. Springer.
- [130] Johannesson, P. and Perjons, E. (2014). *An introduction to design science*, volume 10. Springer.
- [131] Kaasila, J., Ferreira, D., Kostakos, V., and Ojala, T. (2012). Testdroid: automated remote ui testing on android. In *Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia*, pages 1–4.
- [132] Kabiry, S. and Verlinden, D. (2023). Random gui testing with image-based deep reinforcement learning. Master’s thesis, Open Universiteit.
- [133] Kals, S., Kirda, E., Kruegel, C., and Jovanovic, N. (2006). Secubat: a web vulnerability scanner. In *Proceedings of the 15th international conference on World Wide Web*, pages 247–256.
- [134] Kassab, M., DeFranco, J. F., and Laplante, P. A. (2017). Software testing: The state of the practice. *IEEE Software*, 34(5):46–52.
- [135] Kaur, J. and Ramkumar, K. (2022). The recent trends in cyber security: A review. *Journal of King Saud University-Computer and Information Sciences*, 34(8):5766–5781.
- [136] Kervinen, A., Maunumaa, M., Pääkkönen, T., and Katara, M. (2005). Model-based testing through a gui. In *International Workshop on Formal Approaches to Software Testing*, pages 16–31. Springer.
- [137] Khan, M. K. and Bryce, R. (2022). Android gui test generation with sarsa. In *2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 0487–0493. IEEE.

- [138] Khandagale, H. and Halkarnikar, P. (2010). A novel approach for web page change detection system. *International Journal of Computer Theory and Engineering*, 2(3):364.
- [139] Kim, M. O., Coiera, E., and Magrabi, F. (2017). Problems with health information technology and their effects on care delivery and patient outcomes: a systematic review. *Journal of the American Medical Informatics Association*, 24(2):246–250.
- [140] Kirinuki, H., Tanno, H., and Natsukawa, K. (2019). Color: correct locator recommender for broken test scripts using various clues in web application. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 310–320. IEEE.
- [141] Kitchenham, B., Charters, S., et al. (2007). Guidelines for performing systematic literature reviews in software engineering.
- [142] Kitchenham, B. A. (1989). Software quality assurance. *Microprocessors and microsystems*, 13(6):373–381.
- [143] Kitchenham, B. A., Budgen, D., and Brereton, O. P. (2011). Using mapping studies as the basis for further research—a participant-observer case study. *Information and Software Technology*, 53(6):638–651.
- [144] Kong, P., Li, L., Gao, J., Liu, K., Bissyandé, T. F., and Klein, J. (2018). Automated testing of android apps: A systematic literature review. *IEEE Transactions on Reliability*, 68(1):45–66.
- [145] Korn, P., Martínez Normand, L., Pluke, M., Snow-Weaver, A., and Vanderheiden, G. (2013). Guidance on applying WCAG 2.0 to non-web information and communications technologies (WCAG2ICT). Editors’ Draft, work in progress.
- [146] Koroglu, Y. and Sen, A. (2021). Functional test generation from ui test scenarios using reinforcement learning for android applications. *Software Testing, Verification and Reliability*, 31(3):e1752.
- [147] Kousaridas, A., Parisis, G., and Apostolopoulos, T. (2008). An open financial services architecture based on the use of intelligent mobile devices. *Electronic Commerce Research and Applications*, 7(2):232–246.
- [148] Kwon, H., Nam, H., Lee, S., Hahn, C., and Hur, J. (2019). (in-) security of cookies in https: Cookie theft by removing cookie flags. *IEEE Transactions on Information Forensics and Security*, 15:1204–1215.
- [149] Lasi, H., Fettke, P., Kemper, H.-G., Feld, T., and Hoffmann, M. (2014). Industry 4.0. *Business & information systems engineering*, 6:239–242.
- [150] Lelli, V., Blouin, A., and Baudry, B. (2015). Classifying and qualifying gui defects. In *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*, pages 1–10. IEEE.
- [151] Leng, J., Sha, W., Wang, B., Zheng, P., Zhuang, C., Liu, Q., Wuest, T., Mourtzis, D., and Wang, L. (2022). Industry 5.0: Prospect and retrospect. *Journal of Manufacturing Systems*, 65:279–295.

- [152] Leotta, M., Clerissi, D., Ricca, F., and Tonella, P. (2013). Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 272–281. IEEE.
- [153] Li, M., Wang, J., and Damata, L. (2009). Tao project: An intuitive application ui test toolset. In *2009 Sixth International Conference on Information Technology: New Generations*, pages 796–800. IEEE.
- [154] Li, X., Chang, N., Wang, Y., Huang, H., Pei, Y., Wang, L., and Li, X. (2017a). Atom: Automatic maintenance of gui test scripts for evolving mobile applications. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 161–171. IEEE.
- [155] Li, Y., Yang, Z., Guo, Y., and Chen, X. (2017b). Droidbot: a lightweight ui-guided test input generator for android. In *2017 IEEE/ACM 39th Int. Conf. on Software Engineering Companion (ICSE-C)*, pages 23–26. IEEE.
- [156] Li, Y., Yang, Z., Guo, Y., and Chen, X. (2019). Humanoid: A deep learning-based approach to automated black-box android app testing. In *34th ASE*, pages 1070–1073.
- [157] Liebel, G., Alégroth, E., and Feldt, R. (2013). State-of-practice in gui-based system and acceptance testing: An industrial multiple-case study. In *2013 39th Euromicro Conference on Software Engineering and Advanced Applications*, pages 17–24. IEEE.
- [158] Liu, G., Cai, M., Zhao, L., Qin, T., Brown, A., Bischoff, J., and Liu, T.-Y. (2022). Inspector: Pixel-based automated game testing via exploration, detection, and investigation. In *CoG*, pages 237–244. IEEE.
- [159] Liu, L., Tang, W., Buttler, D., and Pu, C. (2002). Information monitoring on the web: a scalable solution. *World Wide Web*, 5:263–304.
- [160] Liu, Z., Chen, C., Wang, J., Chen, M., Wu, B., Che, X., Wang, D., and Wang, Q. (2024). Make llm a testing expert: Bringing human-like interaction to mobile gui testing via functionality-aware decisions. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.
- [161] Liu, Z., Chen, C., Wang, J., Huang, Y., Hu, J., and Wang, Q. (2020). Owl eyes: Spotting ui display issues via visual understanding. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 398–409.
- [162] Magrabi, F., Ong, M.-S., Runciman, W., and Coiera, E. (2012). Using fda reports to inform a classification for health information technology safety problems. *Journal of the American Medical Informatics Association*, 19(1):45–53.
- [163] Mahesh, B. (2020). Machine learning algorithms-a review. *International Journal of Science and Research (IJSR)*. [Internet], 9(1):381–386.
- [164] Manès, V. J., Han, H., Han, C., Cha, S. K., Egele, M., Schwartz, E. J., and Woo, M. (2019). The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331.

- [165] Mao, K., Harman, M., and Jia, Y. (2016). Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 94–105.
- [166] Marchetto, A., Tiella, R., Tonella, P., Alshahwan, N., and Harman, M. (2011). Crawlability metrics for automated web testing. *International journal on software tools for technology transfer*, 13(2):131–149.
- [167] Marchetto, A., Tonella, P., and Ricca, F. (2008). State-based testing of ajax web applications. In *2008 1st international conference on software testing, verification, and validation*, pages 121–130. IEEE.
- [168] Mariani, L., Pezze, M., Riganelli, O., and Santoro, M. (2012). Autoblacktest: Automatic black-box testing of interactive applications. In *2012 IEEE fifth international conference on software testing, verification and validation*, pages 81–90. IEEE.
- [169] Mariani, L., Pezzè, M., Riganelli, O., and Santoro, M. (2014). Automatic testing of gui-based applications. *Software Testing, Verification and Reliability*, 24(5):341–366.
- [170] Mariani, L., Pezzè, M., and Zuddas, D. (2018). Augusto: Exploiting popular functionalities for the generation of semantic gui tests with oracles. In *Proceedings of the 40th International Conference on Software Engineering*, pages 280–290.
- [171] Marín, B., Vos, T. E., Paiva, A. C., Fasolino, A. R., Snoeck, M., et al. (2022). Enactest-european innovation alliance for testing education. In *Proceedings of RCIS 2022 Workshops and Research Projects Track co-located with the 16th International Conference on Research Challenges in Information Science*.
- [172] Martínez, M., Esparcia, A. I., Rueda, U., Vos, T. E., and Ortega, C. (2016). Automated localisation testing in industry with test. In *Testing Software and Systems: 28th IFIP WG 6.1 International Conference, ICTSS 2016, Graz, Austria, October 17-19, 2016, Proceedings 28*, pages 241–248. Springer.
- [173] Martínez, M., Esparcia-Alcázar, A. I., Vos, T. E., Aho, P., and i Cors, J. F. (2018). Towards automated testing of the internet of things: Results obtained with the testar tool. In *Leveraging Applications of Formal Methods, Verification and Validation. Distributed Systems: 8th International Symposium, ISO LA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part III 8*, pages 375–385. Springer.
- [174] Memon, A., Banerjee, I., and Nagarajan, A. (2003a). What test oracle should i use for effective gui testing? In *18th IEEE Int. Conf. on Automated Software Engineering, 2003*, pages 164–173. IEEE.
- [175] Memon, A., Banerjee, I., Nguyen, B. N., and Robbins, B. (2013). The first decade of gui ripping: Extensions, applications, and broader impacts. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 11–20. IEEE.
- [176] Memon, A. M., Banerjee, I., and Nagarajan, A. (2003b). Gui ripping: reverse engineering of graphical user interfaces for testing. In *WCRE*, volume 3, page 260.

- [177] Memon, A. M., Soffa, M. L., and Pollack, M. E. (2001). Coverage criteria for gui testing. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 256–267.
- [178] Menting, T. (2022). Extending testar’s capabilities by integrating ocr for detecting textual presentation failures. Master’s thesis, Open Universiteit.
- [179] Mesbah, A. and Prasad, M. R. (2011). Automated cross-browser compatibility testing. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 561–570.
- [180] Mesbah, A. and Van Deursen, A. (2009). Invariant-based automatic testing of ajax user interfaces. In *31st ICSE*, pages 210–220. IEEE.
- [181] Mesbah, A., Van Deursen, A., and Lenselink, S. (2012). Crawling ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (TWEB)*, 6(1):1–30.
- [182] Meyer, M. (2014). Continuous integration and its tools. *IEEE software*, 31(3):14–16.
- [183] Moran, K., Linares-Vásquez, M., Bernal-Cárdenas, C., Vendome, C., and Poshyvanyk, D. (2017). Crashescope: A practical tool for automated testing of android applications. In *Int. Conf. on Software Engineering Companion*, pages 15–18. IEEE.
- [184] Moran, K., Watson, C., Hoskins, J., Purnell, G., and Poshyvanyk, D. (2018). Detecting and summarizing gui changes in evolving mobile apps. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 543–553.
- [185] Moreira, R. M., Paiva, A. C., Nabuco, M., and Memon, A. (2017). Pattern-based gui testing: Bridging the gap between design and quality assurance. *Software Testing, Verification and Reliability*, 27(3):e1629.
- [186] Moura, T. S. d., Alves, E. L., Figueirêdo, H. F. d., and Baptista, C. d. S. (2023). Cytestion: Automated gui testing for web applications. In *Proceedings of the XXXVII Brazilian Symposium on Software Engineering*, pages 388–397.
- [187] Mozgovoy, M. (2011). Dependency-based rules for grammar checking with languagetool. In *2011 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 209–212. IEEE.
- [188] Mulders, A. (2020). Inferring state models in testar. Master’s thesis, Open Universiteit.
- [189] Mulders, A., Valdes, O. R., Ricós, F. P., Aho, P., Marín, B., and Vos, T. E. (2022). State model inference through the gui using run-time test generation. In *International Conference on Research Challenges in Information Science*, pages 546–563. Springer.
- [190] Muller, T. and Knoll, A. (2009). Virtualization techniques for cross platform automated software builds, tests and deployment. In *2009 Fourth International Conference on Software Engineering Advances*, pages 73–77. IEEE.

- [191] Murphy, A., Laurent, T., and Ventresque, A. (2022). The case for grammatical evolution in test generation. In *GECCO*, pages 1946–1947. ACM.
- [192] Musen, M. A., Middleton, B., and Greenes, R. A. (2021). Clinical decision-support systems. In *Biomedical informatics: computer applications in health care and biomedicine*, pages 795–840. Springer.
- [193] Myers, G. J., Sandler, C., and Badgett, T. (2011). *The art of software testing*. John Wiley & Sons.
- [194] Nafis Fuad, M. and Sakib, K. (2022). ebat: An efficient automated web application testing approach based on tester’s behavior. In *29th APSEC*, pages 482–486.
- [195] Nass, M., Alégroth, E., and Feldt, R. (2021). Why many challenges with gui test automation (will) remain. *Information and Software Technology*, 138:106625.
- [196] Nass, M., Alégroth, E., Feldt, R., and Coppola, R. (2023a). Robust web element identification for evolving applications by considering visual overlaps. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 258–268. IEEE.
- [197] Nass, M., Alégroth, E., Feldt, R., Leotta, M., and Ricca, F. (2023b). Similarity-based web element localization for robust test automation. *ACM Transactions on Software Engineering and Methodology*, 32(3):1–30.
- [198] NetBeans (2009, 2024). Rachota - straightforward timetracking. Last accessed: 3 Sep 2024.
- [199] Nguyen, B. N., Robbins, B., Banerjee, I., and Memon, A. (2014). Guitar: an innovative tool for automated testing of gui-driven software. *Automated software engineering*, 21:65–105.
- [200] Nguyen, V., To, T., and Diep, G.-H. (2021). Generating and selecting resilient and maintainable locators for web automated testing. *Software Testing, Verification and Reliability*, 31(3):e1760.
- [201] North, D. et al. (2006). Introducing bdd. *Better Software*, 12:7.
- [202] Nyman, N. (2000). Using monkey test tools. *Soft. Testing and Quality Eng.*
- [203] OBS-Project (2012, 2023). Free and open source software for live streaming and screen recording. Last accessed: 9 Dec 2022.
- [204] O’Connor, R. V., Elger, P., and Clarke, P. M. (2017). Continuous software engineering—a microservices architecture perspective. *Journal of Software: Evolution and Process*, 29(11):e1866.
- [205] Odukoya, O. K., Stone, J. A., and Chui, M. A. (2014). E-prescribing errors in community pharmacies: exploring consequences and contributing factors. *International journal of medical informatics*, 83(6):427–437.
- [206] Pan, M., Huang, A., Wang, G., Zhang, T., and Li, X. (2020a). Reinforcement learning based curiosity-driven testing of android applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 153–164.

- [207] Pan, M., Xu, T., Pei, Y., Li, Z., Zhang, T., and Li, X. (2020b). Gui-guided test script repair for mobile apps. *IEEE Transactions on Software Engineering*, 48(3):910–929.
- [208] Panthaplackel, S., Nie, P., Gligoric, M., Li, J. J., and Mooney, R. (2020). Learning to update natural language comments based on code changes. In Jurafsky, D., Chai, J., Schluter, N., and Tetreault, J., editors, *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 1853–1868, Online. Association for Computational Linguistics.
- [209] Pantiuchina, J., Mondini, M., Khanna, D., Wang, X., and Abrahamsson, P. (2017). Are software startups applying agile practices? the state of the practice from a large survey. In *Agile Processes in Software Engineering and Extreme Programming: 18th International Conference, XP 2017, Cologne, Germany, May 22-26, 2017, Proceedings 18*, pages 167–183. Springer International Publishing.
- [210] Paramasivam, S. and Leela Velusamy, R. (2023). Quality of service aware routing in software defined video streaming: a survey. *Peer-to-Peer Networking and Applications*, 16(4):1739–1760.
- [211] Parasoft (2017, 2022). The parabank demo application from parasoft. Last accessed: 7 Apr 2022.
- [212] Pastor Ricos, F. (2017). Reconocimiento de widgets automático para aplicaciones java/swing en testar. Bachelor’s thesis, Universitat Politècnica de València.
- [213] Pastor Ricós, F., Doorn, N., Marín, B., and Vos, T. (2024a). Bridging the gap between industry and academia and raising awareness of software testing bias through enactest. In *11th ACM Celebration of Women in Computing: womENCourage*.
- [214] Pastor Ricós, F., Marín, B., Vos, T. E., Neeft, R., and Aho, P. (2024b). Delta gui change detection using inferred models. *Computer Standards & Interfaces*, page 103925.
- [215] Pastor Ricós, F. (2022). Scriptless testing for extended reality systems. In *International Conference on Research Challenges in Information Science*, pages 786–794. Springer.
- [216] Pastor Ricós, F., Marín, B., Prasetya, I., Vos, T. E., Davidson, J., and Hovorka, K. (2024). An industrial experience leveraging the iv4xr framework for bdd testing of a 3d sandbox game. In *International Conference on Research Challenges in Information Science*, pages 393–409. Springer.
- [217] Pehlivan, Z., Ben-Saad, M., and Gançarski, S. (2010). Vi-diff: Understanding web pages changes. In *Database and Expert Systems Applications: 21st International Conference, DEXA 2010, Bilbao, Spain, August 30-September 3, 2010, Proceedings, Part I 21*, pages 1–15. Springer.
- [218] Pereira, L., Sharp, H., de Souza, C., Oliveira, G., Marczak, S., and Bastos, R. (2018). Behavior-driven development benefits and challenges: reports from an industrial study. In *ICASD Companion*, pages 1–4.
- [219] Pezze, M., Rondena, P., and Zuddas, D. (2018). Automatic gui testing of desktop applications: an empirical assessment of the state of the art. In *Companion proceedings for the issta/ecoop 2018 workshops*, pages 54–62.

- [220] Pezzè, M., Rondena, P., and Zuddas, D. (2018). Automatic gui testing of desktop applications: An empirical assessment of the state of the art. In *Companion Proceedings for the ISSTA/ECOOOP 2018 Workshops*, page 54–62. Association for Computing Machinery.
- [221] Politowski, C., Petrillo, F., and Guéhéneuc, Y.-G. (2021). A survey of video game testing. In *2nd AST*, pages 90–99. IEEE.
- [222] Potter, B. and McGraw, G. (2004). Software security testing. *IEEE Security & Privacy*, 2(5):81–85.
- [223] Prada, R., Prasetya, I., Kifetew, F., Dignum, F., Vos, T. E., Lander, J., Donnart, J.-y., Kazmierowski, A., Davidson, J., and Fernandes, P. M. (2020). Agent-based testing of extended reality systems. In *13th ICST*, pages 414–417. IEEE.
- [224] Prancutè, R. (2021). Web of science (wos) and scopus: The titans of bibliographic information in today’s academic world. *Publications*, 9(1):12.
- [225] Prasetya, I., Dastani, M., Prada, R., Vos, T. E. J., Dignum, F., and Kifetew, F. (2020a). Aplib: Tactical agents for testing computer games. In *8th EMAS*, pages 21–41.
- [226] Prasetya, I., Pastor Ricós, F., Kifetew, F. M., Prandi, D., Shirzadehhajimahmood, S., Vos, T. E., Paska, P., Hovorka, K., Ferdous, R., Susi, A., et al. (2022). An agent-based approach to automated game testing: an experience report. In *Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation*, pages 1–8.
- [227] Prasetya, I., Prada, R., Vos, T. E., Kifetew, F., Dignum, F., Lander, J., Donnart, J.-Y., Kazmierowski, A., Davidson, J., and Ricós, F. P. (2020b). Iv4xr-intelligent verification/validation for extended reality based systems. In *14th International Conference on Research Challenges in Information Science (RCIS), Limassol, Cyprus, September 23-25, 2020*, volume 385, pages 647–649. Springer.
- [228] Qi, X.-F., Hua, Y.-L., Wang, P., and Wang, Z.-Y. (2019). Leveraging keyword-guided exploration to build test models for web applications. *Information and Software Technology*, 111:110–119.
- [229] Qiang, M., Miyazaki, S., and Tanaka, K. (2001). Webscan: Discovering and notifying important changes of web sites. In *Database and Expert Systems Applications: 12th International Conference, DEXA 2001 Munich, Germany, September 3–5, 2001 Proceedings 12*, pages 587–598. Springer.
- [230] Rafi, D. M., Moses, K. R. K., Petersen, K., and Mäntylä, M. V. (2012). Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *2012 7th international workshop on automation of software test (AST)*, pages 36–42. IEEE.
- [231] Raina, S. and Agarwal, A. P. (2013). An automated tool for regression testing in web applications. *ACM SIGSOFT Software Engineering Notes*, 38(4):1–4.
- [232] Ralph, P. and Tempero, E. (2018). Construct validity in software engineering research and software metrics. In *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*, pages 13–23.

- [233] Ramler, R., Wetzlmaier, T., and Hoschek, R. (2018). Gui scalability issues of windows desktop applications and how to find them. In *Companion Proceedings for the ISSTA/ECOOOP 2018 Workshops*, pages 63–67.
- [234] Ran, D., Li, Z., Liu, C., Wang, W., Meng, W., Wu, X., Jin, H., Cui, J., Tang, X., and Xie, T. (2022). Automated visual testing for mobile apps in an industrial setting. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, pages 55–64.
- [235] Ren, Y., Gu, Y., Ma, Z., Zhu, H., and Yin, F. (2022). Cross-device difference detector for mobile application gui compatibility testing. In *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 253–260. IEEE.
- [236] Richardson, L. and Ruby, S. (2008). *RESTful web services*. " O'Reilly Media, Inc."
- [237] Ricós, F. P., Aho, P., Vos, T., Boigues, I. T., Blasco, E. C., and Martínez, H. M. (2020). Deploying testar to enable remote testing in an industrial ci pipeline: a case-based evaluation. In *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles: 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20–30, 2020, Proceedings, Part I 9*, pages 543–557. Springer.
- [238] Ricós, F. P., Marín, B., Vos, T., Davidson, J., and Hovorka, K. (2024). Scriptless testing for an industrial 3d sandbox game. In *Proceedings of the 19th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, pages 51–62. INSTICC, SciTePress.
- [239] Ricós, F. P., Neeft, R., Marín, B., Vos, T. E., and Aho, P. (2023a). Using gui change detection for delta testing. In *International Conference on Research Challenges in Information Science*, pages 509–517. Springer.
- [240] Ricós, F. P., Slomp, A., Marín, B., Aho, P., and Vos, T. E. (2023b). Distributed state model inference for scriptless gui testing. *Journal of Systems and Software*, 200:111645.
- [241] Rodríguez-Valdés, O., Vos, T. E., Aho, P., and Marín, B. (2021). 30 years of automated gui testing: a bibliometric analysis. In *International Conference on the Quality of Information and Communications Technology*, pages 473–488. Springer.
- [242] Rodríguez-Valdés, O., Vos, T. E., Marín, B., and Aho, P. (2023). Reinforcement learning for scriptless testing: An empirical investigation of reward functions. In *International Conference on Research Challenges in Information Science*, pages 136–153. Springer.
- [243] Roest, D., Mesbah, A., and Van Deursen, A. (2010). Regression testing ajax applications: Coping with dynamism. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 127–136. IEEE.
- [244] Romdhana, A., Ceccato, M., Georgiu, G. C., Merlo, A., and Tonella, P. (2021). Cosmo: Code coverage made easier for android. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 417–423. IEEE.

- [245] Ross, D. and Gondrom, T. (2013). Rfc 7034: Http header field x-frame-options.
- [246] Rueda, U. U., Esparcia-Alcazar, A., and Vos, T. (2016). Visualization of automated test results obtained by the testar tool. In *XIX Ibero-American Conference on Software Engineering (CIBSE)*, pages 53–66. Curran.
- [247] Rueda, U. U. and Prasetya, W. W. (2016). Automated testing at the user interface level. In *ICT. OPEN2016*.
- [248] Rzig, D. E., Iqbal, N., Attisano, I., Qin, X., and Hassan, F. (2023). Virtual reality (vr) automated testing in the wild: A case study on unity-based vr applications. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1269–1281.
- [249] Saadatmand, M., Enoiu, E. P., Schlingloff, H., Felderer, M., and Afzal, W. (2022). Smartdelta: automated quality assurance and optimization in incremental industrial software systems development. In *2022 25th Euromicro Conference on Digital System Design (DSD)*, pages 754–760. IEEE.
- [250] Sadowski, C., Söderberg, E., Church, L., Sipko, M., and Bacchelli, A. (2018). Modern code review: a case study at google. In *Proceedings of the 40th international conference on software engineering: Software engineering in practice*, pages 181–190.
- [251] Santos, J., Alencar da Costa, D., and Kulesza, U. (2022). Investigating the impact of continuous integration practices on the productivity and quality of open-source projects. In *Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 137–147.
- [252] Selenium (2022). Docker images for the selenium grid server. Last accessed: 7 Apr 2022.
- [253] Shao, F., Xu, R., Haque, W., Xu, J., Zhang, Y., Yang, W., Ye, Y., and Xiao, X. (2021). Webevo: taming web application evolution via detecting semantic structure changes. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 16–28.
- [254] Sharma, R. (2014). Quantitative analysis of automation and manual testing. *International journal of engineering and innovative technology*, 4(1).
- [255] Shylesh, S. (2017). A study of software development life cycle process models. In *National Conference on Reinventing Opportunities in Management, IT, and Social Sciences*, pages 534–541.
- [256] Sillito, J. and Kutomi, E. (2020). Failures and fixes: A study of software system incident response. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 185–195. IEEE.
- [257] Singh, S. K. and Singh, A. (2012). *Software testing*. Vandana Publications.
- [258] Sjøberg, D. I. and Bergersen, G. R. (2022). Construct validity in software engineering. *IEEE TSE*, 49(3):1374–1396.

- [259] Smart, J. and Molak, J. (2023). *BDD in Action*. Simon and Schuster.
- [260] Smith, R. (2007). An overview of the tesseract ocr engine. In *Ninth international conference on document analysis and recognition (ICDAR 2007)*, volume 2, pages 629–633. IEEE.
- [261] Snel, S. (2022). Using code instrumentation to improve text input generation for scriptless gui testing. Master’s thesis, Open Universiteit.
- [262] Song, F., Xu, Z., and Xu, F. (2017). An xpath-based approach to reusing test scripts for android applications. In *2017 14th Web Information Systems and Applications Conference (WISA)*, pages 143–148. IEEE.
- [263] Soremekun, E., Pavese, E., Havrikov, N., Grunske, L., and Zeller, A. (2020). Inputs from hell: Learning Input Distributions for Grammar-Based Test Generation. *IEEE TSE*, 48(4):1138–1153.
- [264] Stocco, A., Yandrapally, R., and Mesbah, A. (2018). Visual web test repair. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 503–514.
- [265] Su, T., Meng, G., Chen, Y., Wu, K., Yang, W., Yao, Y., Pu, G., Liu, Y., and Su, Z. (2017a). Guided, stochastic model-based gui testing of android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 245–256.
- [266] Su, T., Meng, G., Chen, Y., Wu, K., Yang, W., Yao, Y., Pu, G., Liu, Y., and Su, Z. (2017b). Guided, stochastic model-based gui testing of android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, page 245–256. Association for Computing Machinery.
- [267] Tanaka, H. (2019). X-brot: prototyping of compatibility testing tool for web application based on document analysis technology. In *2019 International Conference on Document Analysis and Recognition Workshops (ICDARW)*, volume 7, pages 18–21. IEEE.
- [268] Tanaka, H. (2021). Automating web gui compatibility testing using x-brot: Prototyping and field trial. In *Document Analysis and Recognition–ICDAR 2021 Workshops: Lausanne, Switzerland, September 5–10, 2021, Proceedings, Part II 16*, pages 255–267. Springer.
- [269] Tanno, H., Adachi, Y., Yoshimura, Y., Natsukawa, K., and Iwasaki, H. (2020). Region-based detection of essential differences in image-based visual regression testing. *Journal of Information Processing*, 28:268–278.
- [270] Teevan, J., Dumais, S. T., Liebling, D. J., and Hughes, R. L. (2009). Changing how people view changes on the web. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology*, pages 237–246.
- [271] Thummalapenta, S., Sinha, S., Singhanian, N., and Chandra, S. (2012). Automating test automation. In *2012 34th international conference on software engineering (ICSE)*, pages 881–891. IEEE.

- [272] Tian, J. (2005). *Software quality engineering: testing, quality assurance, and quantifiable improvement*. John Wiley & Sons.
- [273] Totschnig, M. (2012, 2023). Android expenses tracking app. Last accessed: 9 Dec 2022.
- [274] van der Brugge, A., Pastor Ricós, F., Aho, P., Marín, B., and Vos, T. E. (2021). Evaluating testar's effectiveness through code coverage. *Actas de las XXV Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2021)*, pages 1–14.
- [275] Vassallo, C., Palomba, F., Bacchelli, A., and Gall, H. C. (2018). Continuous code quality: are we (really) doing that? In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 790–795.
- [276] Vos, T. E., Aho, P., Pastor Ricós, F., Rodriguez-Valdes, O., and Mulders, A. (2021). testar—scriptless testing through graphical user interface. *Software Testing, Verification and Reliability*, 31(3):e1771.
- [277] Vos, T. E., Kruse, P. M., Condori-Fernández, N., Bauersfeld, S., and Wegener, J. (2015). Testar: Tool support for test automation at the user interface level. *International Journal of Information System Modeling and Design (IJISMD)*, 6(3):46–83.
- [278] Vos, T. E., Marín, B., Escalona, M. J., and Marchetto, A. (2012). A methodological framework for evaluating software testing techniques and tools. In *2012 12th international conference on quality software*, pages 230–239. IEEE.
- [279] Vos, T. E., Tonella, P., Prasetya, I. W. B., Kruse, P. M., Shehory, O., Bagnato, A., and Harman, M. (2014). The fittest tool suite for testing future internet applications. In *Future Internet Testing: First International Workshop, FITTEST 2013, Istanbul, Turkey, November 12, 2013, Revised Selected Papers 1*, pages 1–31. Springer.
- [280] Vos, T. E., Tonella, P., Wegener, J., Harman, M., Prasetya, W., Puoskari, E., and Nir-Buchbinder, Y. (2011). Future internet testing with fittest. In *2011 15th European Conference on Software Maintenance and Reengineering*, pages 355–358. IEEE.
- [281] Vos, T. E. J. and van Vugt-Hage, N. (2019). *Software Testing*. Open Universiteit The Netherlands, Heerlen.
- [282] Vuong, T. A. T. and Takada, S. (2018). A reinforcement learning based approach to automated testing of android applications. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, pages 31–37.
- [283] Walker, A. M. (2017). Tacit knowledge. *Eur. J. Epidemiol.*, 32(4):261–267.
- [284] Wallace, D. R. and Kuhn, D. R. (2001). Failure modes in medical device software: an analysis of 15 years of recall data. *International Journal of Reliability, Quality and Safety Engineering*, 8(04):351–371.
- [285] Walsh, T. A., Kapfhammer, G. M., and McMinn, P. (2020). Automatically identifying potential regressions in the layout of responsive web pages. *Software Testing, Verification and Reliability*, 30(6):e1748.

- [286] Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., and Wang, Q. (2024). Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*.
- [287] Wang, J., Jiang, Y., Xu, C., Cao, C., Ma, X., and Lu, J. (2020). Combodroid: generating high-quality test inputs for android apps via use case combinations. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 469–480.
- [288] Wang, W., Sampath, S., Lei, Y., Kacker, R., Kuhn, R., and Lawrence, J. (2016). Using combinatorial testing to build navigation graphs for dynamic web applications. *Software Testing, Verification and Reliability*, 26(4):318–346.
- [289] Wang, X. (2022). Vrtest: an extensible framework for automatic testing of virtual reality scenes. In *ACM/IEEE 44th ICSE Companion*, pages 232–236.
- [290] Watada, J., Roy, A., Kadikar, R., Pham, H., and Xu, B. (2019). Emerging trends, techniques and open issues of containerization: a review. *IEEE Access*, 7:152443–152472.
- [291] Wen, H.-L., Lin, C.-H., Hsieh, T.-H., and Yang, C.-Z. (2015). Pats: A parallel gui testing framework for android applications. In *2015 IEEE 39th annual computer software and applications conference*, volume 2, pages 210–215. IEEE.
- [292] Wetzlmaier, T., Ramler, R., and Putschögl, W. (2016). A framework for monkey gui testing. In *2016 IEEE international conference on software testing, verification and validation (ICST)*, pages 416–423. IEEE.
- [293] Wieringa, R. and Daneva, M. (2015). Six strategies for generalizing software engineering theories. *Science of computer programming*, 101:136–152.
- [294] Wiklund, K., Eldh, S., Sundmark, D., and Lundqvist, K. (2017). Impediments for software test automation: A systematic literature review. *Software Testing, Verification and Reliability*, 27(8):e1639.
- [295] Williams, L. D. (2021). Concepts of digital economy and industry 4.0 in intelligent and information systems. *International Journal of Intelligent Networks*, 2:122–129.
- [296] Wohlin, C. (2014). Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, pages 1–10.
- [297] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in software engineering*. Springer Science & Business Media.
- [298] Woo, D. and Mori, J. (2004). Accessibility: A tool for usability evaluation. In *Computer Human Interaction: 6th Asia Pacific Conference, APCHI 2004, Rotorua, New Zealand, June 29-July 2, 2004. Proceedings 6*, pages 531–539. Springer.
- [299] Xie, M., Feng, S., Xing, Z., Chen, J., and Chen, C. (2020). Uied: a hybrid tool for gui element detection. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1655–1659.

- [300] Xiong, Y., Xu, M., Su, T., Sun, J., Wang, J., Wen, H., Pu, G., He, J., and Su, Z. (2023). An empirical study of functional bugs in android apps. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1319–1331.
- [301] X.Org (2022). X.org and x11 protocol, server, driver, library, and application development. Last accessed: 7 Apr 2022.
- [302] Xu, T., Pan, M., Pei, Y., Li, G., Zeng, X., Zhang, T., Deng, Y., and Li, X. (2021). Guider: Gui structure and vision co-guided test script repair for android apps. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 191–203.
- [303] Yamin, M. (2018). It applications in healthcare management: a survey. *International Journal of Information Technology*, 10(4):503–509.
- [304] Yu, Y., Wang, H., Yin, G., and Wang, T. (2016). Reviewer recommendation for pull-requests in github: What can we learn from code review and bug assignment? *Information and Software Technology*, 74:204–218.
- [305] Zen, M. and Vanderdonckt, J. (2014). Towards an evaluation of graphical user interfaces aesthetics based on metrics. In *2014 IEEE Eighth International Conference on Research Challenges in Information Science (RCIS)*, pages 1–12. IEEE.
- [306] Zhang, S., Lü, H., and Ernst, M. D. (2013). Automatically repairing broken workflows for evolving gui applications. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 45–55.
- [307] Zhao, Z.-Q., Zheng, P., Xu, S.-t., and Wu, X. (2019). Object detection with deep learning: A review. *IEEE transactions on neural networks and learning systems*, 30(11):3212–3232.
- [308] Zheng, Y., Xie, X., Su, T., Ma, L., Hao, J., Meng, Z., Liu, Y., Shen, R., Chen, Y., and Fan, C. (2019). Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning. In *34th ASE*, pages 772–784. IEEE.
- [309] Zhu, H., Hall, P. A., and May, J. H. (1997). Software unit test coverage and adequacy. *Acm computing surveys (csur)*, 29(4):366–427.

