A Model-Based Approach to Mobile Application Testing

Weidong Xu
School of Computer Science and Engineering
Xi'an Technological University
Xi'an, 710021, China
E-mail: xuweidong@st.xatu.edu.cn

Jing Cheng
School of Computer Science and Engineering
Xi'an Technological University
Xi'an, 710021, China
E-mail: chengjing@xatu.edu.cn

***Abstract*—Modeling the automated testing of mobile applications is a crucial aspect of mobile application automation testing. Due to the varied styles and complex interactions of mobile applications, automated modeling methods are urgently required, particularly in the context of their short development cycles, large numbers, and fast version iterations and updates. In this paper, we address the challenge of exploring mobile application behavior and state based on robotic testing environment without invading the application interior, and propose a method for automated exploration of GUI components and GUI events of applications combined with application domain knowledge to generate mobile application GUI semantic test models. Our results show that the proposed semantic model achieves 70.6% and 82.4% defect detection rate in the robot vision environment and simulation environment, respectively. Compared with the comparative testing method that can only find application crash defects, our method can explore both crash defects and functional anomalies with the application semantic understanding and domain knowledge, thereby extending the automated mobile application functional testing capability of mobile applications. In response to the limitations of mobile application automated testing modeling mentioned above, this paper introduces an automated testing method based on semantic models. It uses the proposed semantic testing model to guide the purposeful exploration of the tested application's states. Subsequently, it generates positive and negative test cases based on the domain knowledge associated with the semantic model. This modeling approach leverages domain models in the mobile application field to conduct automated modeling tests imbued with functional significance, guided by domain knowledge. This optimization aims to address the shortcomings of current automated testing, particularly in terms of model reuse and test expansion.**

Keywords-Mobile Application; Semantic Testing Model; Automated Testing; Test Coverage

I. INTRODUCTION

With the gradual replacement of traditional desktop software by mobile applications as the mainstream software tools in people's daily work, study and life, the development of mobile applications introduces complex new features that make quality assurance more challenging compared to desktop software [1]-[8]. However, the growth in testing needs is in contrast to the limited availability of testing tools and testers. Currently, both industry and academia are increasingly emphasizing the adoption and exploration of automated testing techniques to address the testing needs of mobile applications [9]-[12].

Model-based automated testing is a widely studied testing method that involves constructing test models by mining the state and behavior of the application, and subsequently utilizing these models to generate test cases [13]. Model-based automated testing typically involves completing static modeling of mobile applications based on GUI and dynamic modeling based on behavior jumping, and describing mobile application behavior using inter-state relationship models such as Finite State Machine (FSM). For instance, GUI Ripper [14] facilitates automated exploration modeling of applications, while tools like UI Automator [15]-[16] are utilized to obtain the GUI tree of an application and target the selection of events to complete exploration modeling of mobile applications.

Research on improvements related to model based automated testing: on the one hand, to achieve automatic evolution of the model, e.g., Gu et al [17] can find differences and quickly achieve model evolution after application version update. DeltaDroid [18] builds a defect model that can generate new cases under different conditions based on existing test cases combined with actual GUI and system actions to detect dynamic installation defects in Android applications. On the other hand, the modeling is guided by enhanced application knowledge, e.g., MEGDroid [19] uses model abstraction and model-to-model migration methods to achieve accurate generation of application events; Pan et al [20]-[23] use manual construction of richer test models to guide automated testing.

To overcome the limitations of mobile application automation test modeling mentioned above, this chapter proposes a semantic model-based automation test approach. This approach fuses the mobile application domain model to achieve a semantic matching association between mobile application GUI state and domain knowledge. This facilitates the automated generation of a mobile application GUI semantic test model, which is subsequently used to verify the testing effectiveness of the modeling method.

II. Mobile application semantic testing model

When it comes to mobile application testing, a typical model-based approach involves modeling the relationship between mobile application GUI states and GUI events to establish a series of jumps triggered by events between different GUI states in the application. One example of such an approach is the finite state machine (FSM) model [24]. However, current model-based approaches for mobile application testing are limited to direct records of GUI states and GUI events. These models can only describe the jump-trigger relationship between different GUI states of the application, without understanding the functional meaning of the application. To overcome the aforementioned problems, this paper proposes a method that integrates semantic ontology models with traditional GUI testing models. This method involves building a semantic testing model for mobile applications by extracting semantic information from the GUI of mobile applications and attaching semantic concepts to GUI states and GUI events.

Definition 1: A typical semantic definition of OWL, describing ontology with a formal definition of six tuples:

$$onto\log y = \left\{C, A_C, R, A^R, H, X\right\} \qquad (1)$$

A^c denotes the set of attributes of each concept, the set of concept attributes $A^c(c_i)$, each concept c_i in the set of concepts C is used to represent a set of objects of the same kind and can be described by the same set of attributes.

R denotes the set of relations between concepts, relation r_i (c_p,c_q) that is, each relation r_i in relation R represents a binary relation between concepts cp and cq, and an instance of this relation is a pair of concept objects (c_p,c_q).

A^R denotes the set of attributes of each relation, and the set of relation attributes $A^R(r_i)$ is used to represent the attributes of relation $r_{i.}$

H represents a concept hierarchy, where it is a hierarchy of sets of concepts denoted as C. H also includes a set of parent-child relations that exist between the concepts in C.

The set of axioms is represented as X, where each axiom within X serves as a constraint on the attribute values of a concept and its relation, or as a constraint on the relations between conceptual objects.

Definition 2: The mobile application domain metamodel is defined as a 4-tuple $OAPP = \left\{C, I, R, X\right\}$.

The set of mobile application ontology concepts is denoted by C and is composed of three subsets: entity, action, and task. Each subset contains a concept identifier, concept type, and semantic name.

I represents a collection of instances of mobile application ontology concepts. These instances are concrete textual representations of mobile application component values that are recognized knowledge in the domain. For instance, examples of instances for the entity "place name" could include "Xi'an", "Beijing", and "Shanghai". These instances can be used as the origin or destination in an air service application.

R denotes the set of semantic relations of the mobile application, which contains the inter-concept relation R_{cc} and the concept-instance relation R_{ci} .

X is the constraint definition of the mobile application domain model, which encompasses inter-concept relationship constraints, constraints on concept attributes, instance data constraints, and more. For instance, one constraint could be that a task must consist of at least one entity and one action, and that the instance of the entity "place name" can only be described by text.

Definition 3: The domain model action flow graph is defined as a 5-tuple, $D=\{A,T,F,\alpha_S,\alpha_f\}$.

Action state set A: finite set of mobile application interaction actions.

Action flow control set T: control set of mobile application action sequences.

set of action flow relations F: set of relations from one action state to another action state.

Initial states: a_s, $a_f \in A$.

End state: a_f, $a_f \in A$.

$A=\{a_1,a_2,a_3,\ldots,a_n\}$ is the set of action states, which is the set of mobile application test interaction actions. Each action state is connected by the control flow T in series to form an action sequence. The action state set includes the initial state a_s , the end state a_f , the intermediate states $\{a_m | m=1,2,3,\ldots,n\}$.

$T=\{t_1,t_2,t_3,\ldots,t_n\}$ is the action flow control set, which is the control set of the mobile application test action sequence.

$F=\{f_1,f_2,f_3,\ldots,f_n\}$, $F\subseteq(A\times T)\cup(T\times A)$ is the set of action relations describing the combination of action states and action flow controls.

$a_S=\{a\in A | (a,t)\in F\}$ For the initial state only backward control flow t, $a_f=\{a\in A | (t,a)\in F\}$ for the end state only forward control flow t, $a_m=\{a\in A | (a,t)\in F\wedge(t,a)\in F\}$ must have both forward and backward control flows.

Definition 4: A component in a GUI that cannot be split is an atomic component. The basic components of a GUI in general are atomic components, such as text buttons (Button), text (Text View), images (Image View), etc. AC denotes the set of atomic components in a GUI, $\forall ac\in AC$, $ac=\{ac^t,ac^v,ac^a,ac^s,ac^p\}$, where:

ac^t denotes the component type of the atomic component ac.

ac^v indicates the value of the atomic component ac, which is an optional attribute.

ac^a indicates the possible actions of the atomic component ac, e.g. a button is usually bound to a click action.

ac^s denotes the semantics of the atomic component ac. The semantic entity of ac^v is obtained by mapping ac^v to the domain model ac^s: $ac^v\to\{C_e | C_e\in C_E\}$.

Definition 5: A component composed of atomic components in a GUI is a semantic composite component, e.g., ListView, ToolBar, etc. CC denotes a collection of semantic composite components in a GUI, $cc=\{cc^{ac},cc^t,cc^a,cc^s,cc^p\}$, where:

cc^{ac} denotes the composition of the semantic composite component cc, i.e., which atomic components the semantic composite component cc is composed of, cc^{ac} is the set of atomic components, $cc^{ac}\in AC$.

The component type of a semantic composite component is denoted by cct, and this attribute is optional. It's possible that there may not be a corresponding GUI component type for the semantic composite component.

The semantics of the semantic composite component cc is represented by λc^{cs}, which is determined by the relationship in the domain model where the semantics of the constituent atomic components reside. This can be denoted $\wedge ac^s\to cc^s$. It should be noted that despite being a composite component, the semantics of cc still corresponds to the entities present in the domain model.

Definition 6: The extended semantic FSM model, FSM-ES, is an extension of the typical FSM model that uses the 5-tuple setting. However, it introduces a semantic extension to the expression of the state-hopping relationship: $FSM-ES=\{S,\Sigma,\delta,S_0,F\}$.

The infinite non-empty state set of the GUI is denoted as S, which encompasses all possible states of the application being tested. For ∀s∈S, $s=\{AC,CC,S^S\}$, where AC represents the GUI atomic component, CC represents the GUI semantic composite component, and S^S denotes the semantics of the GUI state that is being represented by the GUI component.

δ is the state transfer function that maps S × Σ to the transition function of S δ:S × ∑→S. ∀s ∈ S, ∀e ∈ ∑.

The notation δ (s, e) refers to the set of states that can be accessed by transitioning from the GUI state s through event e.

III. A MODEL-BASED APPROACH TO MOBILE APPLICATION TESTING

We propose a semantic model-based mobile application testing method, which consists of two parts: visual semantic model-driven GUI modeling and task subgraph-based test case generation.

In the realm of semantic model-driven automated test modeling, the following critical modules are present:

A. FSM-ES model building

The process of semantic model-driven GUI modeling is illustrated in Algorithm 1, which takes the mobile application domain model (ADM), the generic model (GDM), and the application under test (AUT) as inputs and produces the FSM-ES model of the application under test as outputs. The following pseudo code outlines this process:

Algorithm 1

Input. Application under test AUT, application domain model ADM, generic model GDM

Output. Application test model FSM-ES

```
1. while true do
2.    Get the current GUI gui_current of the application under test
3.    Perform visual recognition of GUI component elements on gui_current to get GUI component information gui_info
4.    Match the gui_info of the current GUI with the ADM for semantic similarity to get the vector gui_vc
5.    gui_action inferAction(gui_vector)
6.    Get the response interface vector gui_vr
7.    if gui_vc differs from gui_vr then
8.       mark gui_a in gui_action as executed
9. Generate a path to "gui_vc, gui_a, gui_vr" f
10.      if path f does not exist in fsm_es then
11. Path f is added to fsm_es
12.      end if
13.   else
14.      Logging exceptions
15.   end if
16.      if there is no unexecuted action in gui_action or the exploration timeout then
17.         break
18.      end if
19. end while
20. return
21.
22. function inferAction(gui_vector)
23.    for each gui state in ADM do
24.       if gui state == gui_vector then
25.          Reasoning generates gui actions in the domain corresponding to the gui state and saves them to gui_action
26.       else
26.          Generate executable actions for the interface based on GDM probabilities and save them to gui_action
28.       end if
29.    end for
30.    return gui_action
31. end function
```

The primary objective of the FSM-ES model is to explore the attainable states of the application and associate each state with the task ontology outlined by the domain model. The structure of the FSM-ES model is depicted in TABLE I. FSM-ES semantic structure of music playback application Cathay Pacific

Area	Mssion	State Collection	Action Semantic Set
Music playback	Register	{S_0,S_1,S_2,S_3,S_4,S_{12}}	{select login, agree to the agreement, enter the username, enter the password, and click 1ogin}
	Basin information	{S_5,S_6,S_7,S_8,S_9}	{Recently played, locally downloaded personal cloud drive, friend 1ist, favorite play1ist}
	Discovering Music	{S_{10},S_{11},S_{12}}	{Dai1y recommendation, click 1ike, c1ick play}
	Search Services	{S_{13},S_{14},S_{15},S_{16}}	{Click to type, c1ick to search, clear search,1isten to music and recognize music}
	Persoral Settings	{S_{17},S_{18},S_{19},S_{20},S_{21},S_{22},S_{23}}	{Message center, personal privacy, personalized services, advanced settings, about, 1og out, switch accounts}

B. Task Subgraph Generation

The FSM-ES model is utilized to map to the action flow diagram of the application domain model, which generates task subgraphs for the functional tasks of the application to produce test cases.

If every action concept in the task concept, as defined in the FSM-ES model, corresponds to an action state found in an AFG in the domain, then the task concept is deemed to comply with a specific action flow graph (AFG). The following pseudo code outlines this process:

Algorithm 2

```
Input.    Application Semantic Model FSM-ES,
Domain Model Action Flow Graph AFG
Output.    Task subgraph TFG
1. for each task in the semantic model    do
2.    for each task task in each path f do
3.       task_path     generatePath(task, AFG)
4.       if task_path has a serial relationship
5.          with TFG then
5.          Generate TFG by storing task_
6.          path in sequence
6.       else
7.          return exception and interrupt location
8.          break
9.       end if
10. end for
11.end for
12.return TFG
13.function generatePath(task, AFG)
14.  if task contains the action state and AFG
meets rule 2 then
15.     Generate a feasible action flow based on
AFG inference action
16.     task_path sets the continuity flag
17.  else
18.     task_path sets the end flag
19.  end if
20.  return task_path
21. end function
```

As observed, the domain knowledge incorporated in the action flow diagram can be leveraged to supplement the unexplored inter-state action behavior, thereby generating test judgment criteria for mobile applications.

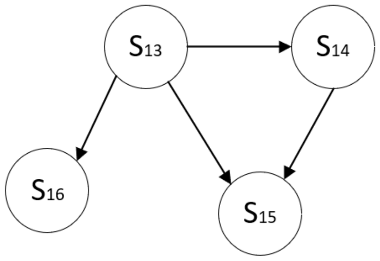

Figure 1. Task states explored by FSM-ES

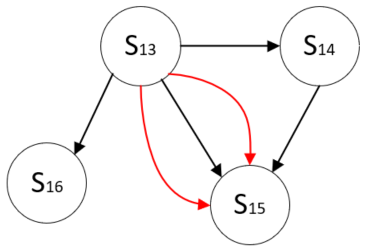

Figure 2. Task Subgraph of Domain Action Flowchart

Upon examining the actual GUI of NetEase cloud music's search song task in (f), it becomes apparent that while the FSM-ES has explored the GUI states, it has failed to account for the iterative behaviors of keying and deleting in S_{13} state and S_{15} state. This is because the FSM-ES prioritizes state exploration over other factors. However, by extending the corresponding action flow diagram definition in the domain ontology library based on domain knowledge, a pathway can be generated, and this domain knowledge can be effectively applied to GUI modeling.

The process of generating task subgraphs from FSM-ES is essentially the instantiation of the mobile application domain model on the application under test. This transforms the abstract concept relationships of the mobile application domain model into a concrete sequence of application action behaviors, making the mobile application testable for execution.

C. Test case generation based on task subgraph

Test case generation comprises two main components: test sequence generation and test data generation. Test sequences are generated by defining semantics-oriented test coverage criteria to guide the traversal of task subgraphs.

Coverage decision rule 1: Existence decision c(A, B), i.e., if the element in A exists in B, the coverage decision is satisfied.

Coverage decision rule 2: Sequential decision s(A, B), i.e., if the elements in A are sequential and conform to the sequential arrangement in B, the coverage decision is satisfied.

Coverage rule 3: Key point decision d (A, B), i.e., if there is an element in A that matches the key element in B, then the coverage decision is satisfied.

Definition a: Semantic concept entity coverage criteria

The set of test cases ensures coverage of both the conceptual entities present in the application domain model being tested and all GUI states formed by these entities.

$$EntityCoverage = c\left(F_A, O_A\right) \cdot s\left(T_G, F_G\right) \qquad (2)$$

Where:

$c(F_A\ ,\ O_A)$ — the coverage of the conceptual entity FA involved in the FSM-ES model of the application under test with the entity OA included in the domain model.

$s(T_G\ ,\ F_G)$ — the coverage of all GUI states involved in the test case set with the GUI states contained in the FSM-ES model of the application under test.

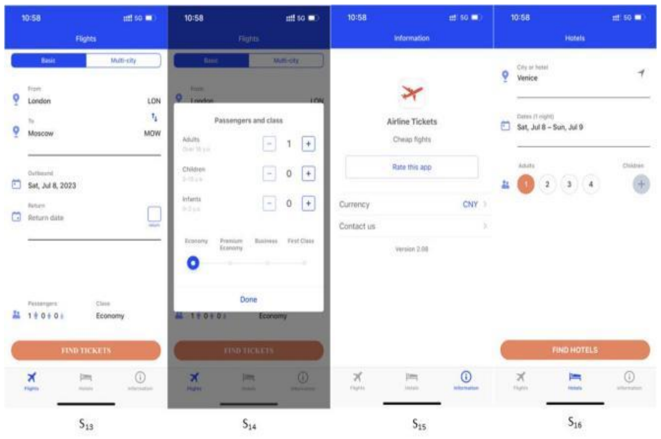

Figure 3. Login Action Test Case

Definition b: Semantic concept action coverage criteria

Test cases satisfy the coverage of actions involved in the action flow diagram of the application domain model being tested. Action coverage is evaluated independently for each action flow diagram. The coverage of semantic concept actions is calculated using the equation (3).

$$ActionCoverage = \frac{\sum_{m=1}^{M} c\left(S_m \cdot D_m\right)}{M} \qquad (3)$$

M—M is the number of task subgraphs included in the FSM-ES model of the application under test, and the action sequence coverage within a subtask is calculated for each task subgraph.

$c\left(S_m \cdot D_m\right)$ —The set of action sequences involved in the set of test cases of the subtask Sm and the coverage of feasible action sequences contained in the activity diagram of the subtaskv D_m.

Definition c: Semantic concept task coverage criteria

Test cases fulfill the coverage requirements of the subgraph of the task being tested, effectively covering all relationships between GUI states. The evaluation of coverage for semantic concept tasks is calculated using the equation presented in (4).

$$TaskCoverage = \frac{d\left(X, X_D\right)}{M} \quad (4)$$

X — the number of application subtasks covered by the test case set

X_D — the number of tasks defined in the domain model

The coverage of semantic concept tasks, which focuses on the application's functional completeness, is calculated by assessing the coverage of tasks in the domain model using the test case set. The number of subtasks covered by the test case set is represented by X, and subtask coverage is assessed using the d-judgment rule. Specifically, if a test case can cover any pathway from the initial state to the end state of a task subgraph, it is deemed as covered and assigned a value of 1. Conversely, if there is no pathway from the initial state to the end state of the task subgraph, it is regarded as not covered and assigned a value of 0.

$$Cov = \alpha \cdot EntityCov + \beta \cdot Action + \gamma \cdot TaskCov \quad (5)$$

Where, α、β、γ correspond to the adjustable parameter weights of the three coverage criteria.

a. To generate test cases based on task subgraphs, it is necessary to cover as many feasible paths as possible to achieve high test case coverage and optimize testing effectiveness.

Algorithm 3

```
Input.  Task subgraph TFG of the application under test
Output. Test case sequence TCi, i=1,2,3,4,......
1. while there is an untraversed path in tfg do
2.     node← getStartNode(tfg)
3.     Create a new sequence TCi with node number i at the beginning
4.     while node is not the end node with degree 0 do
5.         if node has not been traversed then
6.             p← generatePath(node, tfg) //generate test behavior path
7.             Add the node and events from p to TCi
8.             Set the corresponding path in tfg to traversed state
9.         else
10.            p ← generatePath(node, tfg) //generate test behavior paths
11.        end if
12.        node ← p.nodef
13.    end while
14.    i++
15. end while
16. return TC
17.
18. function generatePath(node, tfg)
19.    if node has a unique neighboring node and an untraversed path then
20.        Generate this unique path with the node p
21.    else if n ode does not have the same degree of adjacency then
22.        Generate a path with the node that has the highest degree p
23.        else if node has a path that has not been traversed then
24.        Generate p from this untraversed path
25.    else if node has multiple untraversed paths then
26.        if node's neighboring nodes have traversed nodes then
27.            Generate a path p between the node and its traversed neighbors
28.        else
29.            Generate a path from the node to a random neighbor node p
30.        end if
31.    else
32.        Generate a path p between the node and any of the next nodes
33.    end if
34.    return p← <beginning node nodes, event e, ending node nodef>
35. end function
```

For a given task subgraph TSG:

(i) If there exists a unique pathway of the current node with only one neighboring node, the pathway between the current node and that neighboring node is generated as the next test behavior path.

(ii) If the current node has multiple neighboring nodes with different out degrees, the pathway of the current node and the neighboring node with the highest out degree will be generated as the next test behavior path.

(iii) If there are multiple neighboring nodes at the current node and there is an untraversed path between the current node and one of the neighboring nodes, the untraversed path will be generated as the next test behavior path.

(iv) If the current node has multiple neighboring nodes and there are multiple untraversed paths between it and the neighboring nodes, the path of the current node with the traversed neighboring nodes will be generated as the next test behavior path. If there are no traversed neighboring nodes, the path of a random neighboring node will be generated as the next test behavior path.

IV. Experimentation and Analysis

When selecting applications for testing, those with similar functions are grouped together as domain applications, such as airline service applications, file management applications, news applications, and so on. To establish the domain models, two teams, each consisting of five lab personnel, were invited to create two domain models based on the definition of domain models. These models were cross-checked for accuracy and consistency.

A. Evaluation criteria

Test effectiveness is evaluated at three levels (1) the success rate of test action execution, which determines whether the actions in the test script can be executed without error; (2) the success rate of test script execution, which determines whether the test script can be executed in its entirety without encountering any errors; and (3) the success rate of defect discovery, which evaluates whether known defects in the application set can be identified.

B. Experimental Setup

Upon completion of the exploration of the application under test and subsequent building of the semantic test model, the coverage of the semantic test model with respect to the domain model is determined by analyzing the successful matching of the application state and the domain model as recorded by the exploration algorithm. The coverage results of the FSM-ES models created for each domain tested application, along with the corresponding domain model, are presented in TABLE II. The table indicates that the average coverage of entity concept is 89%, while the average coverage of action concept is also 89%. The average coverage of task concept is 81%. It is true that the non-intrusive environment may have some impact on the modeling process, particularly with regard to factors such as GUI recognition accuracy.

Application crash defects:

ActivityNotFoundException, Activity not found exception.

IllegalArgumentException, illegal parameter exception.

IllegalStateException, illegal state exception.

NullPointer Exception, null pointer exception.

TABLE I. Distribution of defects discovered by various methods

Error type	Defect type found	Semantic Modeling in Robot Vision Environment	Semantic modeling in a simulated environment	Humanoid	Stoat
Application crash defects	ActivityNotFoundException	1	1	3	2
	lllegalArgumentException	3	2	1	1
	lllegalState Exception	3	2	2	2
	NullPointer Exception	4	3	2	0
	OutOfMemoryError	2	2	1	2
	amount	13	10	9	7

TABLE II. DEFECT DISCOVERY RESULTS FOR EACH APPLICATION

APP	Semantic Modeling in Robot Vision Environment			Semantic modeling in a simulated environment		
	Entity Coverage	Action Coverage	Task Coverage	Entity Coverage	Action Coverage	Task Coverage
Apple Music	84%	82%	87%	84%	82%	89%
QQ Music	83%	89%	85%	86%	89%	83%
Music	88%	87%	92%	89%	90%	85%
TunePro Music	93%	92%	93%	93%	92%	93%
Shazam	92%	89%	91%	91%	91%	92%
Spotify	90%	90%	93%	92%	89%	91%
ES File Explorer	89%	87%	87%	89%	86%	89%
average	89%	89%	81%	88%	90%	89%

The button is missing, it should have interacted with a component in the GUI, but the component is not found.

Information is missing; part of the information that should exist is missing.

GUI anomaly, where the GUI changes after interaction, but the GUI interface is displayed differently than it should be.

GUI display anomalies, where GUI display anomalies such as buttons unresponsive or GUI information abnormalities and missing information appear most frequently in the experiment. In addition, only a few exceptions were found for the commercial application, which may be related to the fact that it has been adequately tested, while the open-source application has more defects. No functional anomalies were found for the commercial application in the experiments, only application crashes.

V. CONCLUSIONS

The experiments conducted in this study validate the efficacy of the proposed semantic model-driven automated testing approach. By utilizing the domain semantic model as the core, this approach enhances the reusability of the testing model and introduces a new perspective on mobile application testing. Furthermore, it facilitates a completely non-invasive testing approach, particularly in the robot vision environment. To address the shortcomings of current mobile application functional testing, a semantic model-driven automated testing approach is proposed. The paper investigates an extended semantic model-driven automated testing method, based on a domain model of mobile applications. It first explores the states of the tested application with the goal of achieving maximum reachable states, thereby establishing an extended semantic FSM-ES model. Subseque-ntly, based on the domain model's action flowchart, the FSM-ES model is extended and mapped to a task subgraph with feasible paths as the goal, aiming to cover application functionality. This modeling of the tested application is accomplished from two perspectives: the GUI state reachability relationships (FSM-ES) and feasible paths between GUI states (task subgraph).Following this, by defining semantic coverage-oriented testing criteria, the goal is to achieve the broadest path coverage within the task subgraph. This process generates test cases targeting application functionality. Through testing verification in various application domains such as aviation services, among 13 discovered defect categories totaling 34 defects, the test cases generated by the semantic testing model achieved defect detection rates of 70.6% in the robot's visual environment and 82.4% in a simulated environment. Moreover, the semantic model-generated test cases were able to simultaneously detect application crashes and functional anomalies, supporting complex automated testing of functionalities with strict requirements for behavior sequences and test inputs.

To address the shortcomings of current mobile application functional testing, a semantic model-driven automated testing approach is proposed. The paper investigates an extended semantic model-driven automated testing method, based on a domain model of mobile applications. It first explores the states of the tested application with the goal of achieving maximum reachable states, thereby establishing an extended semantic FSM-ES model. Subsequently, based on the domain model's action flowchart, the FSM-ES model is extended and mapped to a task subgraph with feasible paths as the goal, aiming to cover application functionality. This modeling of the tested application is accomplished from two perspectives: the GUI state reachability relationships (FSM-ES) and feasible paths between GUI states (task subgraph). Following this, by defining semantic coverage-oriented testing criteria, the goal is to achieve the broadest path coverage within the task subgraph. This process generates test cases targeting application functionality. Through testing verification in various application domains such as aviation services, among 13 discovered defect categories totaling 34 defects, the test cases generated by the semantic testing model achieved defect detection rates of 70.6% in the robot's visual environment and 82.4% in a simulated environment. Moreover, the semantic model-generated test cases were able to simultaneously detect application crashes and functional anomalies, supporting complex automated testing of functionalities with strict requirements for behavior sequences and test inputs.

REFERENCES

[1] Tramontana P, Amalfitano D, Amatucci N, et al. Automated functional testing of mobile applications: a systematic mapping study [J]. Software Quality Journal, 2019, 27(1):149-201.

[2] Kong P, Li L, Gao J, et al. Automated testing of android apps: A systematic literature review [J]. IEEE Transactions on Reliability, 2018, 68(1): 45-66.

[3] Cruz L, Abreu R, Lo D. To the attention of mobile software developers: guess what, test your app! [J]. Empirical Software Engineering, 2019, 24(4): 2438-2468.

[4] Wimalasooriya C, Licorish S A, da Costa D A, et al. A systematic mapping study addressing the reliability of mobile applications: The need to move beyond testing reliability [J]. Journal of Systems and Software, 2022, 186: 111166.

[5] Al-Subaihin A A, Sarro F, Black S, et al. App store effects on software engineering practices [J]. IEEE Transactions on Software Engineering, 2019, 47(2): 300-319.

[6] Luo C, Goncalves J, Velloso E, et al. A survey of context simulation for testing mobile context-aware applications [J]. ACM Computing Surveys, 2020, 53(1): 1-39.

[7] Amalfitano D, Amatucci N, Memon A M, et al. A general framework for comparing automatic testing techniques of Android mobile apps [J]. Journal of Systems and Software, 2017, 125(c): 322-343.

[8] Linares-Vásquez M, Bernal-Cárdenas C, Moran K, et al. How do developers test android applications?[C]//2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2017: 613-622.

[9] Li N, Offutt J. Test oracle strategies for model-based testing[J]. IEEE Transactions on Software Engineering, 2016, 43(4): 372-395.

[10] Banerjee I. Advances in model-based testing of GUI-based software[M]//Advances in Computers. Elsevier, 2017, 105: 45-78.

[11] Automator[EB/OL], https://developer.android.com/training/testing/ui-automator, 2020-3. Ngo C D, Pastore F, Briand L. Automated, cost-effective, and update-driven app testing [J], ACM Transactions onSoftware Engineering and Methodology, 2022, 31(4):1-51.

[12] Gu T, Sun C, Ma X, et al. Practical GUI testing of Android applications via model abstraction and refinement[C]//2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 2019: 269-280.

[13] Ghorbani N, Jabbarvand R, Salehnamadi N, et al. DeltaDroid: Dynamic Delivery Testing in Android [J]. ACM Transactions on Software Engineering and Methodology, 2022.

[14] Hasan H, Ladani B T, Zamani B. MEGDroid: A model-driven event generation framework for dynamic android malware analysis [J]. Information and Software Technology, 2021, 135:106569.

[15] Pan M, Lu Y, Pei Y, et al. Effective testing of Android apps using extended IFML models [J]. Journal of Systems and Software, 2020, 159:110433.

[16] Perera A, Aleti A, Böhme M, et al. Defect prediction guided search-based software testing[C]//2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2020:448-460.

[17] Su T, Meng G, Chen Y, et al. Guided, stochastic model-based GUI testing of Android apps [C]//Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. 2017:245-256.

[18] Zhong B, Wu H, Li H, et al. A scientometric analysis and critical review of construction related ontology research [J]. Automation in Construction, 2019, 101:17-31.

[19] Web Ontology Language (OWL)[EB/OL]. https://www.w3.org/OWL, 2012-12/2022-9.

[20] Li Y, Yang Z, Guo Y, et al. Humanoid: a deep learning-based approach to automated black-box Android app testing [C]//2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2019:1070-1073.

[21] Wu X, Sahoo D, Hoi S C H. Recent advances in deep learning for object detection [J]. Neurocomputing, 2020, 396:39-64.

[22] Deka B, Huang Z, Franzen C, et al. Rico: A mobile app dataset for building

[23] Data-driven design applications [C]//Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology. 2017:845-854.

[24] Ioffe S, Szegedy C. Batch normalization: Accelerating deep network training by reducing internal covariate shift [C]//International conference on machine learning. PMLR, 2015:448-456.