Automatic UI Generation Technique for Mobile Applications on Touch-Screen based Smart Phones

YunSik Son[1] and YangSun Lee[2*]

[1]*Dept. of Computer Engineering, Dongguk University*
26 3-Ga Phil-Dong, Jung-Gu, Seoul 100-715, Korea

[2]*Dept. of Computer Engineering, Seokyeong University*
16-1 Jungneung-Dong, Sungbuk-Ku, Seoul 136-704, Korea

sonbug@dongguk.edu, yslee@skuniv.ac.kr
**Corresponding Author: yslee@skuniv.ac.kr*

Abstract

As touch-screen mobile phones pour into the market, demands for reusing existing mobile applications by adding a touch-screen UI are increasing. In order to service an existing application without a touch-screen UI to touch phones, an on-screen keyboard that implements a virtual keyboard on the touch screen is needed. On-screen keyboards are particularly convenient to use on small mobile phone screens because they reduce the number of keys and consequently increase the size of each. Previous on-screen keyboard generation methods generate fixed keyboard layouts that include every key defined by a system, resulting in smaller and inconvenient keys. Further, the recently studied dynamic analysis method analyzes keys used in source codes during run-time to generate on-screen keyboards optimized for applications but it generates too much overhead to control.

In this study, we proposed the touch-screen UI generating technique based on content's source code by static analysis. To generate the UI automatically, we adopt the notion of key set graphs that store the UI status information of applications to identify UI states by referencing the graphs and generating optimized on-screen keyboards. This method uses effective graph data structures to effectively circumvent the overhead issues featured in the dynamic analysis method.

Keywords: *Software Reuse, Human Computer Interaction, Touch-Screen User Interface*

1. Introduction

Given that the mobile application services market encompasses a wide range of mobile platforms, converting applications developed for a specific platform to be reused in other platforms is essential in expanding the client base [1, 2, 3].

The recent mobile phone market features a number of touch phones in which all functions are implemented with a touch screen UI. Touch phones are emerging as a hot-button product in the market because they offer both compactness and diversity of functions [4].

Touch phones can only run applications with a touch screen UI. However, since most existing applications do not feature a touch screen UI, the process of adding a touch screen UI to existing applications is required in order to reuse them in touch phones [5].

The most effective way to add a touch-screen UI to an application with only a key UI is the on-screen keyboard technique, which defines a virtual keyboard on the touch screen and generates key events that correspond to touches on the screen. Since on-screen keyboard are

placed on small mobile phone screens, reducing the number of keys and consequently increasing the size of each key adds to the convenience.

Various studies were conducted as an effort to propose a method for automatically generating on-screen keyboards to add touch-screen UI to existing applications [6, 7]. Although the resulting methods were successful in terms of automation, they generated fixed keyboard layouts that include all keys defined in a system, which isn't very friendly to use.

In addressing such defect, studies on generating on-screen keyboards optimized for applications by analyzing the keys used in source codes were carried out [5, 8]. Although these methods generate on-screen keyboards optimized for applications, they also generate too much overhead in the process.

This study proposes a method for generating on-screen keyboards based on the static source code analysis method by storing the UI state information in graphs, which are referenced during run-time. This method uses effective graph data structures to yield results optimized for UI states and steers clear from the overhead issue pointed out as drawbacks in existing methods. Chapter 2 reviews previous studies on on-screen keyboard generation and source code analysis, Chapter 3 introduces the static analysis method and discusses the on-screen keyboard generation method using static analysis. Chapter 4 verifies whether optimized touch-screen UIs are generated using the proposed method through a series of experiments. Finally, Chapter 5 draws the conclusion and proposes study topics moving forward.

2. Related Studies

2.1. On-Screen Keyboards

The attempt to apply an on-screen keyboard to mobile applications first took place in gaming. The original mobile version of The Magic Thousand-Character Text was converted to be run in iPhone on which it displayed fixed Gameboy-style touch buttons (Figure 1) [6].

(a) Gameboy like touch button (b) Platform level onscreen keyboard

Figure 1. On-Screen Keyboards

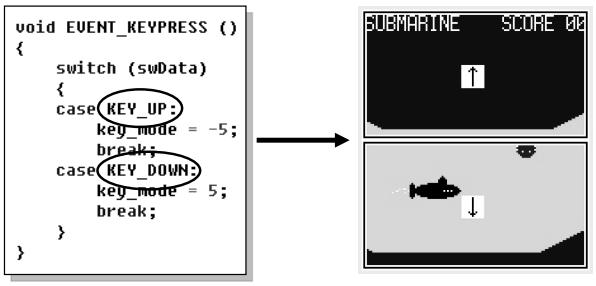
This method allocated a certain area on the screen for touch buttons where being able to apply various designs on buttons is an advantage but not being able to use that area for other purposes in an application is a drawback.

Existing applications can be reused in touch phones without any additional conversion process if on-screen keyboards are offered in the platform level. Qbric mobile phones provide window-type on-screen keyboards that feature various functions in the platform level [7].

However, since this method always displays every key of a system regardless of the key UI of an application, the sizes of the keys are forced to shrink, which causes inconvenient user-experience.

2.2. Code Analysis

A study on generating an on-screen keyboard by analyzing the source code to identify the keys used in the application thereby optimizing the number of keys was conducted in order to overcome the drawbacks of the fixed layout (Figure 2) [5].



K_{active} = { KEY_UP, KEY_DOWN }

Figure 2. Touch-Screen UI Adapted for Application

This method excludes keys unused in an application and thus reduces the number of leys and enlarges the size of each key for convenience.

The UI of a standard mobile application contains multiple UI modules where a particular UI module is run depending on the program's internal status. Here, the state that runs the particular UI module is called the UI state.

The method described above does not take into account that the context of a UI module changes according to the UI state and analyzes only the keys used by an application as a whole thereby failing to remove the keys unnecessary for that application.

The active key set of a UI state refers to a set of keys with functions defined to that particular UI module. By finding the active key set of the current UI state during program execution, we can build a layout consisted of only the keys included in the active key set and display an on-screen keyboard optimized to the UI state.

An active key monitoring method using the dynamic analysis method was proposed based on the above ideas. This method inserts a monitoring code in a program, calls the monitoring code during execution to obtain the active key set, and selects an optimal layout from the set.

While the static analysis method is able to obtain results through a relatively easier means since it obtains results through actual experimentation, there exists the issue of degraded

execution performance or the original program because it has to run the program every time for every result.

The active key monitoring method also requires the key event handler to be repeated, which could cause heavy overhead and sometimes lead to critical faults depending on the system. Further, securing execution performance is even more vital for mobile applications because cell phones to which they are serviced are limited in terms of their performance.

While the dynamic analysis method runs every time to obtain results, the static analysis method is able to apply results obtained from a single run to all cases thereby improving the execution performance [9].

In this study, we investigate the UI state and active key set of applications using graphs based on the static analysis method and using that result, we propose a method of generating a touch-screen UI optimized to the UI state.

3. Static Analysis Method

Control flow analysis, the most general static analysis method, breaks a program down into base block units with unchanging control flows and analyzes the overall flow of the program through base blocks and control flows.

The control flow analysis method generally analyzes source codes after converting them to CFGs (Control Flow Graph). CFGs can simply be generated from the intermediate result of ASTs (Abstract Syntax Tree) as source code information can be stored in each node for later use in the analysis stage [10].

Generally, UI state information is located in the key event handler and in the modules that it calls. Thus, we limit the targets for converting to CFGs to key event handler-related parts rather than the entire source code in order to alleviate the complexity of analysis.

3.1. UI State Information on CFG

Figure 3 shows an example of converting the key event handler of a standard application into a CFG. UI state from a CFG's perspective is a state with control flow defined to execute a particular UI module. Thus, we can deduce two pieces of information by analyzing a CFG.

First, we can look at the conditions that classify the UI state. Conditions that determine the UI state of a program are called UI state conditions and variables included in these conditions are called UI state control variables. We can find the conditions that determine a particular UI state by following the control flow from the CFG to that particular UI module. For example, in Fig. 3, the conditions under which UI module k is executed are ($p = 1$) and ($q \neq r$) where the UI state control variables are p and q.

Second, we can examine the active key set of a particular UI state. A statement that compares the keys pressed by a user appears after following the control flow of a certain UI state where the compared key values become the active keys. All active keys present on the control flow become the active key set. The active key set for the state during which UI module k is run is { KEY_X, KEY_Y, KEY_Z }.

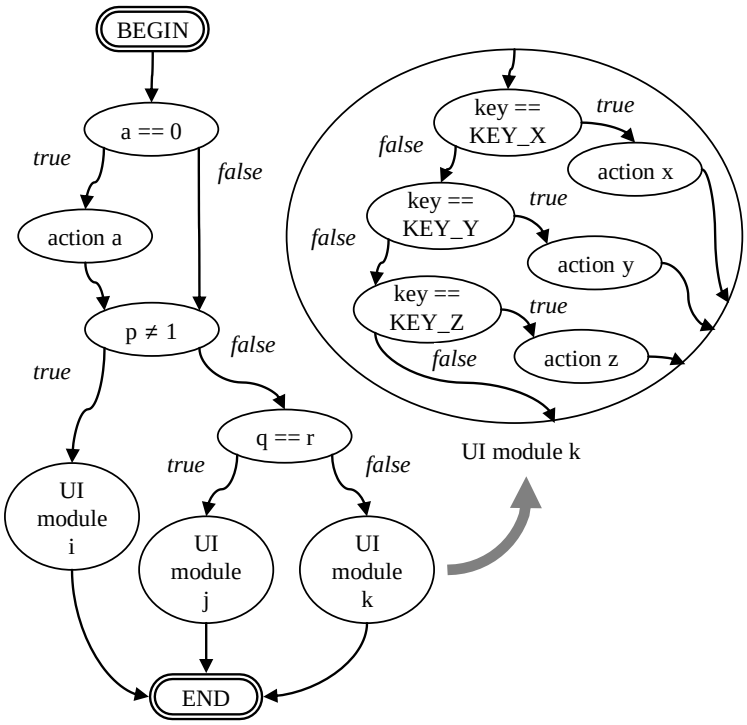


Figure 3. Example CFG of Key Event Hander

3.2. Graph Simplify Algorithms

When analyzing general graphs, removing nodes that have no influence on analysis results will increase the efficiency of graph analysis. This study proposes a graph simplification algorithm that increases the efficiency of graph analysis by removing nodes from CFG that are not relevant to UI state information and adding nodes that store active keys.

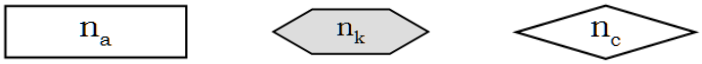
The graph simplification algorithm comprises the following three steps. (1) Node classification, (2) Remove unnecessary node, and (3) Merge adjacent key store node. The simplified graph resulting from running this algorithm is called a KSG (Key Set Graph).

3.2.1 Node Classification: Every node in the graph is traversed and each node is categorized as one of the following three node types in Figure 4.

(1) Action node: All nodes except conditional statements like if and switch are action nodes. All action nodes are removed in the following stage because they have no influence on the UI state.

(2) Key store node: Conditional statement nodes with branches become key store nodes when comparing the values of keys pressed by a user in the logical equation. Key store nodes store the key values in a logical equation in their respective nodes. When there are multiple key values in a logical equation, every one of them are stored. Here, the stored key values represent active keys.

(3) Selection node: Conditional statement nodes with branches that fail to satisfy the conditions of a key store node are all selection nodes.



(1) action node (2) key store node (3) selection node

Figure 4. Symbols of Node Types

Figure 5 is a CFG with node classification applied to the example source code.

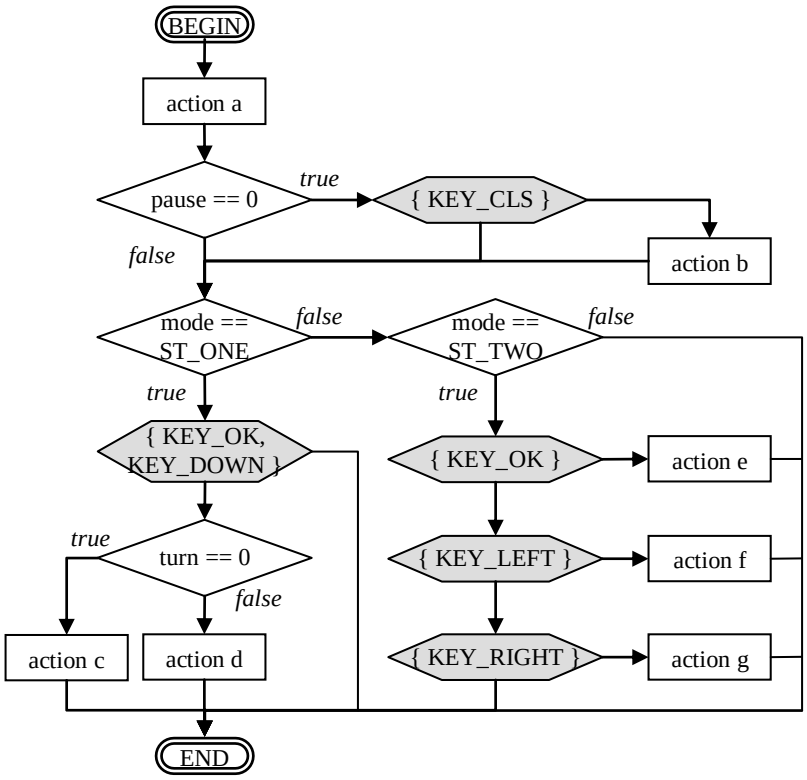


Figure 5. CFG of the Example Source Code

3.2.2. Remove Unnecessary Node: Every node in the graph is traversed during which unnecessary nodes are removed. The node removal algorithm is as shown in Figure 6.

If the node type is active node, the particular node is removed and the in-bound edges of the deleted node are all connected to the out-bound edges of that node.

If the node type is selection node and the two out-bound edges of the node point to the same node, that particular node is removed because it has lost the selection functionality. A node is removed in the same way as the removal of an active node.

Lastly, if the node type is key store node and the two out-bound edges of that node point to the same node, the second edge is removed.

```
REMOVE_UNNECESSARY_NODE( )
for all node n of graph {
    if TYPE(n) = n_a then {
        REMOVE_NODE(n);
    }
    else if TYPE(n) = n_c then {
        n_1 = HEAD_NODE(OUT_EDGE(n, 1));
        n_2 = HEAD_NODE(OUT_EDGE(n, 2));
        if n_1 = n_2 then
            REMOVE_NODE(n)
    }
    else if TYPE(n) = n_k then {
        n_1 = HEAD_NODE(OUT_EDGE(n, 1));
        n_2 = HEAD_NODE(OUT_EDGE(n, 2));
        if n_1 = n_2 then
            DELETE_EDGE(OUT_EDGE(n, 2));
    }
}
```

Figure 6. Unnecessary Node Remove Algorithm

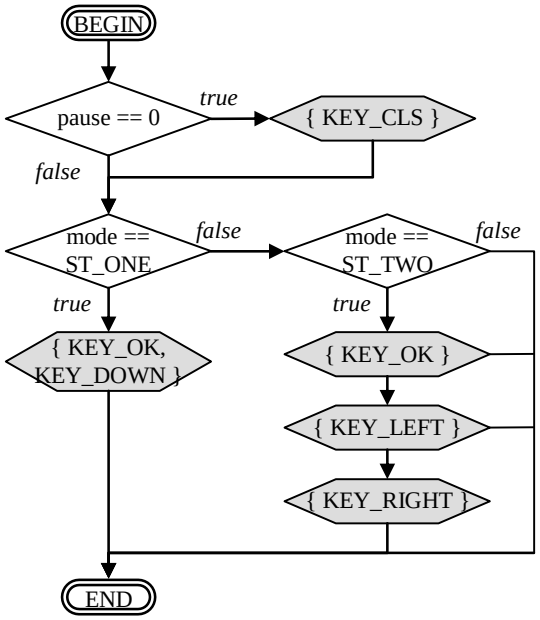


Figure 7. Unnecessary Node Removed Graph

Figure 7 shows the result of having removed the unnecessary nodes from the CFG in Figure 5. All action nodes have been removed and only selection nodes and key store nodes remain in the graph.

3.2.3. Merge Adjacent Key Store Node: In order to increase the graph searching performance, all adjacent key store nodes are combined into one. Here, the active key sets stored in the nodes are also combined. The key store node merging algorithm is as shown in Figure 8

```
MERGE_KEYSTORE_NODE( )
for all node n of graph {
    if TYPE(n) = n_k then {
        n_pre = TAIL_NODE(IN_EDGE(n, 1));
        if TYPE(n_pre) = n_k then {
            ADD_KEY(n_pre, KEY_SET(n));
            REMOVE_NODE (n);
        }
    }
}
```

Figure 8. Key Store Node Merging Algorithm

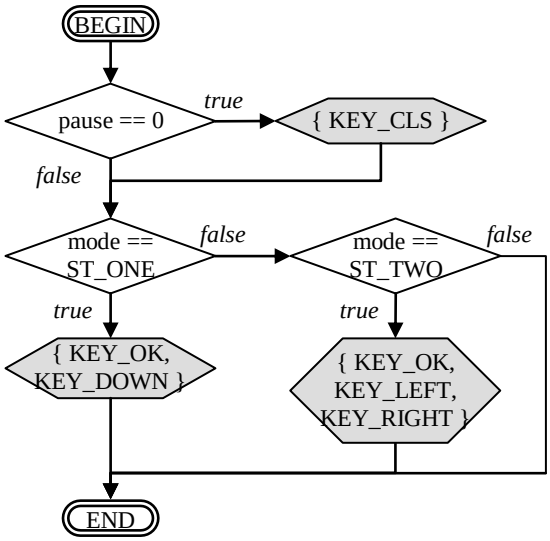


Figure 9. Key-Set Graph of the Example

Figure 9 illustrates the completed KSG, which is created by merging all adjacent key store nodes from the graph of Figure 7.

Selection nodes remaining in the KSG then become the UI state conditions and variables included in the conditional statement of the selection nodes become the UI state control variables. In Figure 9, the UI state control variables are pause and mode.

The KSG is traversed using the values allocated to the state control variables at a certain point during application run-time. When we capture all the keys stored in the key store nodes during the traversal, it becomes the active key set of the current state. If pause = 0 and mode = ST_ONE in Figure 9, the active key set then becomes { KEY_CLS, KEY_OK, KEY_DOWN }.

Here, in order to observe the UI state through the KSG from an arbitrary location, the UI state control variables must be declared globally and their values must not change until the UI state is determined in the key event handler.

3.3. Decision Table

Repetitive graph traversing is required in order to obtain the active key set during program run-time using KSG. Thus, the efficiency of examining the UI state can be increased by creating a decision table using the graph traversing results beforehand and using table lookup in place of graph computation.

A decision table is consisted of UI states categorized by the combinations of UI state conditions in the KSG and the active key set derived from it. Table 1 shows the KSG in Figure 7 represented with a decision table.

Table 1. Decision Table of the Example

UI state condition			**active key set**
pause == 0	**mode == ST_ONE**	**mode == ST_TWO**	
true	*true*	-	{ KEY_CLS, KEY_OK, KEY_DOWN }
	false	*true*	{ KEY_CLS, KEY_OK, KEY_LEFT, KEY_RIGHT }
		false	{ KEY_CLS }
false	*true*	-	{ KEY_OK, KEY_DOWN }
	false	*true*	{ KEY_OK, KEY_LEFT, KEY_RUGHT }
		false	{ }

3.4. Touch-Screen UI Engine

A new application with touch-screen UI function can be created by inserting the touch-screen UI engine code into the source code of an original application and applying the decision table generated by the method described up to this point.

The structure of the new application with touch-screen UI is illustrated in Figure 10. The touch-screen UI engine operates as follows.

(1) UI state checking process: CheckUiState() is executed at regular intervals by using the timer. CheckUiState() monitors the state control variable and once it detects a change in its value, it assumes that the UI state has been changed after which it looks up the decision table to obtain the active key set and execute BuildLayout(). BuildLayout() selects the smallest layout that includes the active key set from the keyboard layout DB provided by the user and sets it as the new layout [5].

(2) Event redirection process: when the user touches the on-screen keyboard, the touch event handler is executed and the RedirectKey() function is called. The RedirectKey() function references the currently-configured layout to generate corresponding key events and execute functions defined in the key event handler.

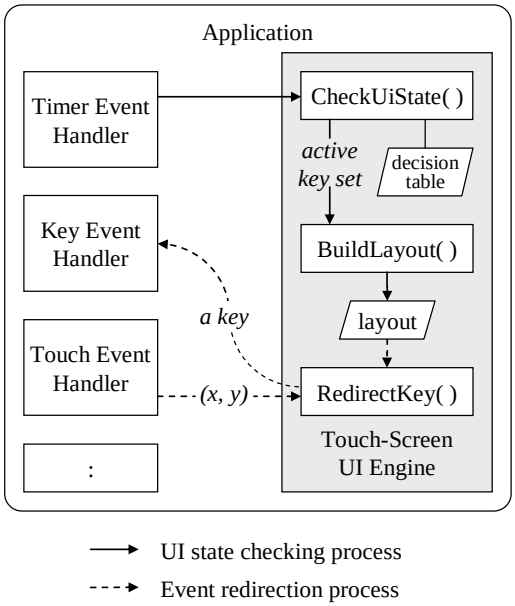


Figure 10. System Model of Touch-Screen UI Engine

4. Experimental Results and Analysis

We apply the proposed static analysis method on the test application to generate a KSG and a decision table and we generate the touch-screen UI-added application and verify the result.

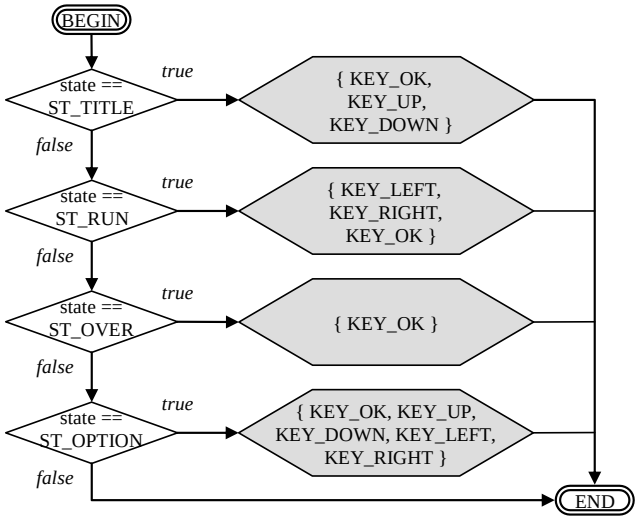
The test environment uses GNEX Emulator that runs in PCs. GNEX is a mobile platform installed in CDMA cell phones through which a large number of applications are serviced. The test application is the SPACE WAR game program written in the MobileC language. Written in a commercial program-like structure, SPACE WAR contains 4 UI states in Table 2.

Table 2. UI Information of SPACE WAR

UI state	Action defined key
title state	{ KEY_OK, KEY_UP, KEY_DOWN }
run state	{ KEY_LEFT, KEY_RIGHT, KEY_OK }
game over state	{ KEY_OK }
option state	{ KEY_OK, KEY_UP, KEY_DOWN, KEY_LEFT, KEY_RIGHT }

In the first experiment, we verify whether the four predefined UI states are clearly distinguished by generating a KSG and a decision table for SPACE WAR.

Figure 11 illustrates the result of generating a KSG and a decision table of SPACE WAR by applying the graph simplification algorithm. When we look at the decision table, we can see that there are five UI states depending on the UI state control variable "state".



(a) Key set graph

UI state condition				**active key set**
state == ST_TITLE	**state == ST_RUN**	**state == ST_OVER**	**state == ST_OPTION**	
true	-	-	-	{ KEY_OK, KEY_UP, KEY_DOWN }
false	*true*	-	-	{ KEY_LEFT, KEY_RUGHT, KEY_OK }
	false	*true*	-	{ KEY_OK }
		false	*true*	{ KEY_OK, KEY_UP, KEY_DOWN, KEY_LEFT, KEY_RIGHT }
			false	{ }

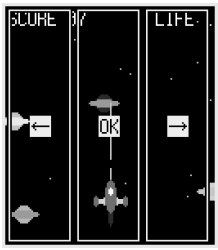
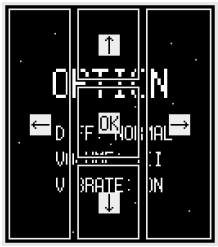
(b) Decision table

Figure 11. Intermediate Result of SPACE WAR

The 5th UI state can exist logically regardless of the purpose of the original program and the active key set becomes $\emptyset$. Therefore, if we exclude the last UI state, we can see that the KSG and the decision table have been created successfully because of the four UI states and that the active key set corresponds perfectly with Table 2.

In the second experiment, we insert the touch-screen UI engine and the decision table of Figure 11 into SPACE WAR to create a new application and we observe the touch screen UI operation of this application.

Table 3. Touch-Screen UI of SPACE WAR

UI state condition	Active key set	Guide display
state == ST_TITLE	{ KEY_OK, KEY_UP, KEY_DOWN }	
state == ST_RUN	{ KEY_LEFT, KEY_RUGHT, KEY_OK }	
state == ST_OVER	{ KEY_OK }	
state == ST_OPTION	{ KEY_OK, KEY_UP, KEY_DOWN, KEY_LEFT, KEY_RIGHT }	

The result of executing the SPACE WAR program with touch-screen UI is as shown in Table 3. The touch screen UI engine has the guide display function, which displays on the screen the layout every time the on-screen keyboard layout is changed. The guide display result of Table 3 shows that the layout optimized to the active key set of each UI has been applied.

5. Conclusions

In this study, we proposed a method of using KSG to store the UI status of applications in and looking up the decision table of KSG to effectively identify the UI status based on the static analysis method.

Since UI state changes are broadcasted promptly through a decision table when adding touch-screen UI to applications by using this method, user-experience can be improved significantly by generating layouts optimized in UI state units. Also, since decision table lookup computations are processed extremely swiftly, the overhead issue pointed out in the dynamic analysis method can be circumvented.

On the other hand, since this method uses the static analysis method, UI status control variables must always be declared global and they cannot be modified within the key event handler.

Further study topics include tracking changes in UI state control variables by adopting the data flow method and automatically generating touch-screen UIs for specific platforms by formalizing the various language and platform-dependent parts.

Acknowledgements

This paper was extended from the previous research paper “Static Analysis Method for Touch-Screen UI Generation on Mobile Application” in International Conference on Computer and Applications 2012.

References

[1] J. Kang and C. Jung, “Mobile programming using WIPI GNEX”, Saengreung Press, **(2006)**.
[2] J. Cho, C. Hong and Y. Lee, "Implementation of Automatic Translation Tools of GVM to BREW Contents in Mobile Environment", Journal of Korea Multimedia Society, vol. 9, **(2005)**, pp. 38-49.
[3] K. Kang, D. Kang and C. Hong, "Development of Conversion Solutions for Interoperability of Applications on Different Mobile Internet Platform", Journal of Korea Contents Society, vol. 7, no. 4, **(2007)**, pp. 1-9.
[4] M. Kim and J. Yoon, "Implementation of Mobile Game Interface through an Operating Interface Analysis of Touchscreen Devices," Journal of Korean Society of Design Science, vol. 22, no. 1, **(2009)**, pp. 231-244.
[5] S. Ko, Y. Son, J. Park and S. Oh, "Automatic Generation Technique of Touch-Screen Interface for Mobile Game Contents", Journal of Korean Institute of Information Scientists and Engineers: Computing Practices and Letters, vol. 15, no. 11, **(2009)**, pp. 866-869.
[6] C. Lee, “The Proposal which gives at the Mobile Game Enterprise and the Case of the Magic Thousand-Character Text Game”, http://blog.dreamwiz.com/chanjin/9646413, **(2009)**.
[7] Electronic Publication: Pantech, iSKY IM-R470S Qbric, http://www.isky.co.kr, **(2009)**.
[8] L. Magnien, J. L. Bouraoui and N. Vigouroux, "Mobile text input with soft keyboards: Optimization by means of visual clues", In Proceedings of the 6th International Symposium of Mobile Human-Computer Interaction, Glasgow, UK, **(2004)**, pp. 337-341.
[9] D. D. Cruz, P. Henriques and J. S. Pinto, "Code Analysis: Past and Present", In Proceedings of the 3rd International Workshop on Foundations and Techniques for Open Source Software Certification, York, UK, **(2009)**.
[10] S. M. Oh, “Introduction to Compilers”, Jungiksa, **(2004)**.
[11] Sinjisoft, “MobileC Programming Guide”, MCC-PRG, **(2009)**, pp. 301-312.
[12] D. Binkley, “Source Code Analysis: A Road Map”, 2007 Future of Software Engineering, **(2007)**, pp. 104-119.
[13] F. Nielson, H. R. Nielson and C. Hankin, “Principles of Program Analysis”, Springer, **(2005)**.
[14] J. Han, Q. Xu and B. Kim, “The Research on Effective Expression of the Touch Screen GUI”, The Korean Society of Illustration Research, vol. 19, **(2009)**, pp. 57-66.
[15] R. Meier, “Professional Android Application Development”, Wiley Publishing, Inc., **(2009)**.

[16] S. Park, H. Kwon, Y. Kim and Y. Lee, "Design and Implementation of GVM-to-MIDP Automatic Translator for Mobile Game Contents", Journal of Game Society, vol. 3, no. 1-2, **(2006)**, pp. 5-12.
[17] Y. Son, S. Oh and Y. Lee, "Design and Implementation of the GNEX C-to-Android Java Converter using a Source-Level Contents Translator", Journal of Korea Multimedia Society, vol. 13, no. 7, **(2010)**, pp. 1051-1061.
[18] S. Oh, S. Yoon, Y. Son and J. Park, "Development of Mobile C-WIPI C Source Converter", Project Report, Industry-Academy Cooperation Foundation of Dongguk Univ., **(2008)**.
[19] Korea Creative Content Agency, "2008 White Paper on Korean Games: Guide to Korean Game Industry and Culture", **(2008)**, pp. 624-643.
[20] Semantic Designs, Inc., "Software Re-Engineering Tools for Automating Design Upgrade", Research and data for Status Report 95-09-0059, **(2006)**.

Authors

Yunsik Son

He received the B.S. degree from the Dept. of Computer Science, Dongguk University, Seoul, Korea, in 2004, and M.S. and Ph.D. degrees from the Dept. of Computer Engineering, Dongguk University, Seoul, Korea in 2006 and 2009, respectively. Currently, he is a Researcher of the Dept. of Computer Science and Engineering, Dongguk University, Seoul, Korea. His research areas include smart system solutions, secure software, programming languages, compiler construction, and mobile/embedded systems.

YangSun Lee

He received the B.S. degree from the Dept. of Computer Science, Dongguk University, Seoul, Korea, in 1985, and M.S. and Ph.D. degrees from Dept. of Computer Engineering, Dongguk University, Seoul, Korea in 1987 and 2003, respectively. He was a Manager of the Computer Center, Seokyeong University from 1996-2000, a Director of Korea Multimedia Society from 2004, a General Director of Korea Multimedia Society from 2005-2006 and a Vice President of Korea Multimedia Society in 2009. Also, he was a Director of Korea Information Processing Society from 2006, and a President of a Society for the Study of Game at Korea Information Processing Society from 2006. And, he was a Director of Smart Developer Association from 2011-2012. Currently, he is a Professor of Dept. of Computer Engineering, Seokyeong University, Seoul, Korea. His research areas include smart system solutions, programming languages, and embedded systems.