JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 34, 1119-1139 (2018)
DOI: 10.6688/JISE.201809_34(5).0002

Coupling Analysis and Visualization of KDT Scripts

CHIEN-HUNG LIU AND WOEI-KAE CHEN
Department of Computer Science and Information Engineering
National Taipei University of Technology
Taipei, 106 Taiwan
E-mail: {cliu; wkchen}@ntut.edu.tw

In keyword-driven testing (KDT), a test case depends on a number of keywords, and a keyword depends on some other keywords or GUI components (widgets). Such dependency is also known as the coupling between test cases, keywords, and components. This paper proposes a coupling measure called unweighted coupling and a visualization tool called KTV (Keyword-driven Test script Visualizer) with which the structure of KDT scripts can be quickly grasped. A case study is conducted to assess the maintainability of KDT scripts with various degrees of couplings. The results indicate that, when maintaining KDT scripts, a low-coupling script required, on average, less changes than a high-coupling one. In addition, the estimated change impact offered by KTV was highly correlated with the actual maintenance cost. Thus, in general, the low-coupling principle holds for KDT scripts and can be important to a tester who needs to constantly develop/maintain KDT scripts.

Keywords: keyword-driven testing, coupling, maintainability, readability, visualization, software data analysis

1. INTRODUCTION

As GUI (Graphical User Interface) is pervasive in all kinds of software applications (Web, mobile, and rich client applications), GUI testing assumes increasing importance nowadays. To automate GUI testing, a tester prepares a test script (or test case), which contains a sequence of actions (events and assertions) that are performed on the components (widgets) of the GUI [1, 2]. The test script is then executed to exercise the GUI and perform verifications.

When developing a test script, Keyword-Driven Testing (KDT) approach (*e.g.*, [3-6]) is widely used. In KDT, a keyword performs one or more actions and a test case is constructed by using a sequence of keywords. In general, a keyword can also use (call) some other keywords, creating a hierarchical keyword structure. For example, Fig. 1 shows a simple KDT test script (called simply KDT script or script hereafter) supported by Robot Framework [3]. The keyword Login sequentially opens a login page, enters an account name into the account widget, enters a password into the password widget, and then submit the login form. In this example, Login keyword uses another keyword Open Login page, which opens a browser with a specified login URL. The advantages of KDT are that test cases are more concise and readable, and keywords are reusable and can be revised to accommodate to the changes in the GUI.

The principle behind KDT is procedural abstraction. The role of a keyword/action is analogous to a C language function/statement. Fig. 2 depicts a call graph in which test

Received August 30, 2017; revised October 7, 2017; accepted November 4, 2017.
Communicated by Don-Lin Yang.

cases use keywords, and keywords in turn use actions to accomplish a test job. Such relationships between test cases, keywords, and actions are also known as the coupling between them. Note that, as shown in Fig. 2, when a keyword *K* uses an action *A* performed on component *C*, *K* is in fact coupled to *C*, not *A*. This is because *A* is not an object and, without the existence of *C*, *A* cannot be used at all (*e.g.*, suppose *K* uses the click event of a button, *K* depends on the button, not on the click event).

Keyword	Action	Argument	Argument
Login	Arguments	${account}	${password}
	Open Login Page		
	Input Text	account	${account}
	Input Text	password	${password}
	Submit Form		
Open Login Page	Open Browser	http://host/login.html	
	Title Should Be	Login Page	

Fig. 1. A simple login test script.

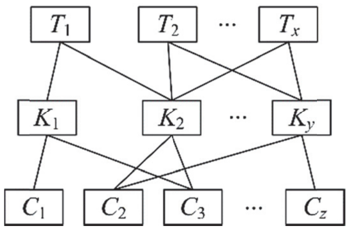


Fig. 2. A general call graph of test cases, keywords, and components. A vertex T_i denotes a test case, a vertex K_i denotes a keyword, and a vertex C_i denotes a component on which an action is performed.

In software design, coupling is a measure of how strongly one software module is connected to, has knowledge of, or relies on other modules [7-9]. It is generally known that a software system designed with low coupling supports low change impact and promotes maintainability. However, to our knowledge, so far there are no researches that study the coupling for the special case of KDT scripts. In particular, it is not even known whether different KDT scripts may have different degrees of couplings. Since the purpose of a KDT script is to create user interactions, it is not the same as a piece of source code that performs intensive arithmetic/logic computations. Thus, code coupling measures or principles may not be directly applicable to the case of KDT scripts.

This paper studies whether coupling influences the maintainability and readability of KDT scripts. This is important to a tester who needs to constantly develop and maintain KDT scripts. A coupling measure called unweighted coupling and a visualization tool called KTV (Keyword-driven Test script Visualizer) are proposed. KTV offers eight different ways of viewing a script so that the tester can quickly grasp the overall structure of the script. In addition, a change impact estimation is offered for the tester to evaluate the cost of changing one or more components. A case study with four experiments is conducted to address whether coupling is related to maintainability and readability and whether the estimated change impact is realistic. The results indicated that (1) given exactly the same sequence of actions to perform, different testers created KDT scripts with completely different couplings; (2) when maintaining KDT scripts, a low-coupling script required, on average, less changes (test cases and keywords) than a high-coupling one; (3) in terms of readability, a low-coupling test script was however not necessarily easier to read than a high-coupling one; and (4) the change impact estimation offered by KTV was highly correlated to the actual maintenance cost. Thus, in general, the results suggest that the low-coupling design principle holds for KDT scripts and keeping the coupling as low as possible can be important to a tester who needs to constantly develop/maintain KDT scripts.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 presents a simple coupling measure for KDT scripts along with coupling visualization and change impact analysis. The design of KTV tool is described in Section 4. Section 5 reports a case study. The concluding remarks and future work are given in Section 6.

2. RELATED WORK

Among various test automation approaches, KDT, offered by Fewster and Graham [4] and Kaner *et al.* [6] among others, has been widely used. KDT is an improvement over data-driven testing. In KDT, both test data and keyword implementations can be taken apart from test scripts and put into external input files. To offer better flexibility, many KDT tools (*e.g.*, [3]) also allow testers to construct high-level keywords. Thus, many testers develop/maintain their own keyword hierarchies by themselves. However, as reported in Section 5.2, a KDT script is not necessarily easy to maintain. The current trend in iterative/agile processes [10] promotes evolutionary refinement of plans, requirements, and design. Thus, the test script associated with the design must also be updated frequently and its maintenance becomes important. That brings up the question, when creating keywords, is there a guideline that a tester should follow? The results of this paper suggest that low coupling is an important design principle for the development of KDT scripts.

Coupling has long been used as an attribute or metric to assess the quality of software. It was first introduced to evaluate the degree of interdependence between modules in structured design development technique [7-9]. The more inter-related (*i.e.*, tightly coupled) the modules are, the more difficult these modules are to understand, change, and reuse. Since then, the concept of coupling has been used to evaluate the quality of software products [11]. A variety of coupling measures, such as the types of coupling [12], quantitative models of coupling [13], information flow metrics [14, 15], and levels of coupling [16], have been proposed to take into account the characteristics of structured design and programming for estimating the qualitative property of modules and determining the coupling algorithmically.

The principle of low coupling has been adapted to object-oriented systems since the features of object-oriented programming languages, such as class, inheritance, and polymorphism, have introduced to a number of new kinds of couplings [17]. Several coupling measurement frameworks particularly for object-oriented systems have been presented. These frameworks identify and measure the coupling of object-oriented systems from different perspectives, such as the coupling between classes [18, 19], coupling related to various types of relationships [20], object level and class level coupling [21], coupling introduced by different types of interactions and coupling directions [22], and coupling extracted from the implementation [23].

The coupling information can be obtained from design documents or from implementation. Thus, coupling measures can be used in different stages of software development process [22]. In addition to the aforementioned researches that have identified and classified various types of coupling and used coupling as an indicator for measuring the "goodness" of a software design or an implementation, coupling measures also have been applied to numerous activities to increase the quality and maintainability of software, such as localizing the error-prone system structure [24], predicting the possible faults [25], analyzing the ripple effects of changes [26], and improving integration testing [27].

As previously described, the measurement of coupling for high-level languages and object-oriented systems has been extensively studied in the literature. However, a KDT script, a procedural abstraction of GUI operations, is not equivalent to a piece of code. Though some KDT tools (*e.g.*, Robot Framework [3]) also support keywords that perform loops, conditional expressions, and even arithmetic/logic computations, the bulk of a script is made of sequences of actions. At the extreme, a test script simply stores all the actions linearly. In case that arithmetic/logic computations are necessary, they are normally encapsulated into special keywords and account for only a very small portion of the script. Therefore, the most prominent coupling in a script is the caller-callee relationships between modules (test cases, keywords, and components). Consequently, general code-coupling measures (*e.g.*, data, stamp, control, common, and content couplings [8]) do not capture the essence of a script nicely. To our knowledge, so far there are no previous researches that specifically address the coupling of KDT scripts. This paper and its earlier work [28] fill this gap by offering a simple measure along with an evaluation.

3. COUPLING ANALYSIS AND VISUALIZATION OF KDT SCRIPTS

This section describes the proposed coupling metric for KDT scripts and illustrates that a script with a bad keyword design can have an increased coupling. Various diagrams for visualizing the define-use relationships among the modules of KDT scripts are presented to facilitate the coupling analysis. Based on the coupling metrics and visualization, the estimation of the possible change impacts of KDT scripts is addressed.

3.1 Coupling Measure

To measure and analyze the coupling of KDT scripts, we define the coupling of a KDT script as the degree to which each module (either a test case, keyword, or component) relies on each one of the other modules. A KDT script is like the source code of a high-level language. However, a typical KDT script contains mainly actions. Its purpose is to drive the GUI and verify its correctness. Therefore, a KDT script does not normally perform complicated arithmetic/logic computations. Consequently, language constructs such as structure and class are unnecessary and are not supported by many popular KDT enabled tools (*e.g.*, [3, 5]). According to Myers [8], there are five different kinds of code couplings (*i.e.*, data, stamp, control, common, and content couplings). However, since a KDT script is not designed for doing computations, using sophisticated code coupling metrics can be unrealistic. Therefore, we focus mainly on evaluating the caller-callee relationships between modules.

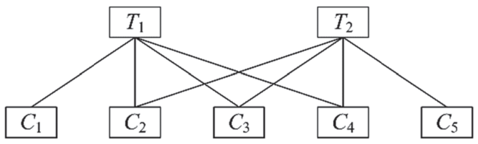
We use a coupling diagram, like Fig. 2, to illustrate the coupling between modules (for simplicity, we use the term keyword to represent a module hereafter). When a keyword uses another keyword, there is a coupling between them. Therefore, a straightforward measure of coupling is to evaluate the density of the coupling diagram – the higher the density, the higher the coupling. More precisely, a coupling diagram is a directed, unweighted graph whose vertices are keywords and there is an edge (K_i, K_j) if and only if K_i uses K_j. Thus, given a coupling diagram $G = (V, E)$ where V is the set of vertices and E the set of edges, we can define *Unweighted Coupling* (*UC*) as:

$$UC = \frac{|E|}{|V|-1}, \quad |V| > 1. \tag{1}$$

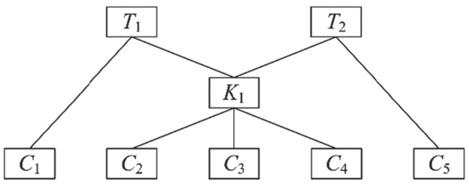
In Eq. (1), $|E|$ is divided by $|V| - 1$ to obtain the density and the value of $|V|$ is greater than 1 because a typical KDT script should have at least one test case with at least one event. Since a normal coupling diagram should have at least $|V| - 1$ edges (*i.e.*, a connected graph without any unused keywords), the minimum value for *UC* is 1. Fig. 3 (a) is an example with 7 vertices (2 test cases and 5 components) and 8 edges (note: as all edge directions are top-down, arrows are not shown in the graph). In this case, $UC = 8/(7-1) = 1.33$. Fig. 3 (b) is another example with $UC = 1$. Note that we did not define *UC* using the standard graph density equation (*i.e.*, $|E|\div(|V|(|V| - 1))$). This is because a coupling diagram is normally not a dense graph and does not have $O(|V|^2)$ edges. If the standard graph density equation is used, the value of *UC* would generally be too small to read.

The *UC* metric can be used to evaluate if KDT scripts have a high coupling which can bring about maintenance problems. How could a KDT script have a high coupling? We present three typical cases that keywords are not properly designed, resulting an increased coupling: (1) using too few keywords, (2) using keywords that are not reusable, and (3) having redundant keywords. The first case appears when many actions are repeatedly executed for several times, yet no keywords are used to simplify the repetition. For example, Fig. 3 (a) shows that both T_1 and T_2 use the same actions from C_2, C_3, and C_4. In this case, by adding a new keyword K_1 (Fig. 3 (b)) to encapsulate these actions, T_1 and T_2 no longer depend directly on C_2, C_3, and C_4. Thus, the overall coupling is reduced from 1.33 to 1.

The second case appears when a keyword cannot be easily reused. Fig. 4 (a) shows an example that T_1 uses K_1 to perform actions on C_1, C_2 and C_3. On the surface, T_2 also performs actions on C_1, C_2 and C_3, and thus, it is maybe possible that T_2 could reuse K_1. However, suppose K_1 contains one or more actions that T_2 does not need (*e.g.*, K_1 uses two different actions of C_1, but T_2 needs only one of them), T_2 is forced to use C_1, C_2, and C_3 directly, resulting the diagram shown in Fig. 4 (a). Note that the edge (K_1, C_1) is labeled with a weight of 2 to indicate that K_1 uses two different actions of C_1. The fundamental problem in this case is that K_1 is not designed as a common keyword for both T_1 and T_2. Thus, a better design is to refactor K_1 into K_1' so that K_1' uses only one action of C_1 and both T_1 and T_2 can use K_1', and such a refactoring also reduces the overall coupling from 1.4 to 1.2. Fig. 4 (b) shows the diagram after the refactoring in which the weight of (K_1', C_1) is reduced to 1. Note that in order to use K_1', T_1 also needs to be changed so that it directly uses one action of C_1. Such a change is indicated in Fig. 4 (b) by adding the edge (T_1, C_1) with weight 1.

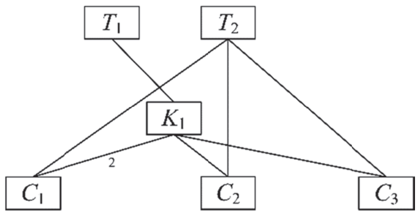


(a) No keywords (UC = 1.33).

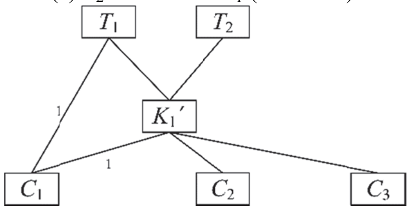


(b) Adding a keyword K_1 (UC = 1).

Fig. 3. The coupling diagram of the first case.



(a) T_2 cannot use K_1 (UC = 1.4).

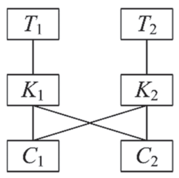


(b) K_1 is redesigned into K_1' (UC = 1.2).

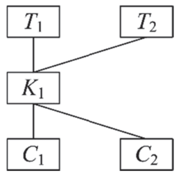
Fig. 4. The coupling diagram of the second case.

The third case appears when a test script grows larger and larger, and the tester is unaware of an existing keyword and creates a new one, or when a keyword is not properly parameterized and cannot be reused. Fig. 5 (a) shows an example that K_1 and K_2 are in fact identical (or identical when the differences between K_1 and K_2 can be eliminated by using parameter substitutions). In this case, T_2 can use K_1 instead of K_2 (Fig. 5 (b)), reducing the overall coupling from 1.2 to 1.

From the above three cases, it can be seen that a different keyword design gives a different coupling. In Section 5, we will further study whether keyword designs could be different and report their couplings.



(a) K_2 is redundant ($UC = 1.2$).



(b) K_2 is removed ($UC = 1$).

Fig. 5. The coupling diagram of the third case.

3.2 Coupling Visualization

A coupling diagram showing the calling relationships between the modules is very useful for the analysis of the coupling in a KDT script. However, as a script evolves and gets larger, the corresponding coupling diagram can suffer from overplotting and, hence, becomes very difficult to read and understand since a large number of modules and complicated coupling relationships are to be displayed. To address this problem, several diagrams are proposed so that the coupling can be visualized hierarchically.

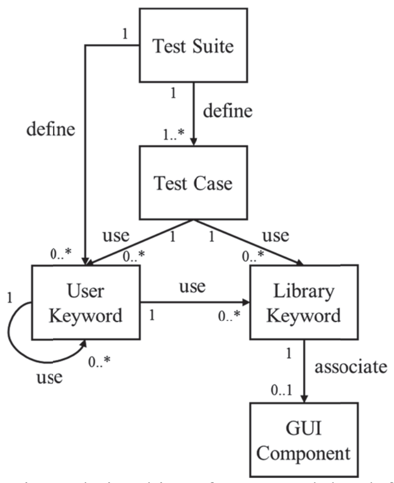


Fig. 6. The define-use-associate relationships of KDT modules defined in Robot Framework.

Fig. 6 shows the define-use-associate relationships among test suites (TS), test cases (TC), user keywords (UK), library keywords (LK), and GUI components (C). Basically, a test suite can define a set of test cases and a set of user (*i.e.*, user-defined) keywords. A user keyword can use some other user keywords and/or library keywords. Each test case is composed of a sequence of keywords that can be either user or library keywords. Each library keyword produces an action performed on a GUI component and is thus associated with the component. Note that library keywords are normally provided by KDT tools and a user keyword cannot directly perform actions on a GUI component without using library keywords.

Based on the relationships shown in Fig. 6, Table 1 lists eight different diagrams used for visualizing the coupling of KDT scripts. These diagrams can show different perspectives of the coupling among a selection of modules in the KDT scripts. For example, the TC-UK diagram shows the coupling between the test cases and user keywords while the TC-UK-C diagram shows the coupling among test cases, user keywords, and components. Each diagram shows only a portion of the entire hierarchical structure of KDT scripts, except for the TS-TC-K-C diagram. As a consequence, the display complexity of the diagram is greatly reduced. With the diagrams, testers can quickly capture the overall structure of KDT scripts and visualize the coupling among selected KDT modules.

In general, when a tester starts reading/understanding a test script, a top-down ordering is the most frequently used reading sequence. In this case, TS is read first, followed by TC, UK, and then LK. The first three diagrams (TS-TC, TC-UK, and UK-LK) support this kind of reading sequence. The next three diagrams (TS-UK, TS-LK, and TC-LK), on the other hand, offer the rest of the relationships indicated in Fig. 6. The last two diagrams (TC-UK-C and TS-TC-K-C) offer an overview of the entire test script in two different levels. In particular, a TS-TC-K-C diagram shows all the modules.

Table 1. Diagrams for visualizing the coupling of KDT scripts.

Diagram name	Showing the coupling among …
TS-TC diagram	test suites and associated test cases
TC-UK diagram	test case and associated user keywords
UK-LK diagram	user keywords and associated library keywords
TS-UK diagram	test suites and associated the user keywords
TS-LK diagram	test suites and associated library keywords
TC-LK diagram	test cases and associated library keywords
TC-UK-C diagram	test cases, user keywords, and components
TS-TC-K-C diagram	all the modules

To illustrate, Fig. 3 (b) is an example of a TC-UK-C diagram and Fig. 7 is an example of a TS-TC-K-C diagram. In Fig. 7, the script contains two test cases, six user keywords, five library keywords, and five components. Note that among the six user keywords, DismissMessageDialog and AssertSavable uses some other user keywords and the rest of the four user keywords use library keywords directly.

Note that Table 1 does not include all possible diagrams for visualizing the coupling relationships among all the modules in KDT scripts. For example, it could be helpful to have the TC-LK-C and TS-TC-LK-C diagrams that show the hierarchical structures of coupling among LK and other modules, such as test suites, test cases and components. Further, it could be nice to have the TS-TC-UK-C diagram rendering the coupling structure of all KDT modules except for LK. However, for simplicity, the current implementation of KTV described in Section 4 does not include the TC-LK-C, TS-TC-LK-C, and TS-TC-UK-C diagrams since the coupling visualization of UK is the main focus of the paper and the TS-TC-K-C diagram can offer an alternative for viewing the coupling structure of TS-TC-UK-C.

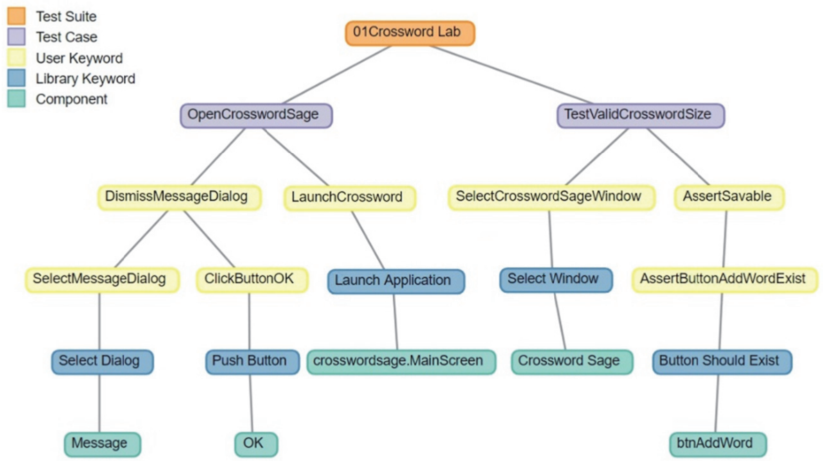


Fig. 7. An example of the TS-TC-K-C diagram.

3.3 Change Impact Analysis

It is not uncommon that some features of a software system are changed during the development and evolution of the system. When a change request is proposed, it is crucial to estimate the possible impact of the change in order to evaluate whether the change request can be approved. As the source code of a program changes, so does the corresponding KDT script. When a GUI component is changed, the coupling of the script can be used to analyze the possible change impact. The idea is to compute the "ripple effect" of changes based on the coupling dependencies. A module *m* can be affected by a module *n* if *m* uses (*i.e.*, depends on) *n*. Thus, depending on the impact level (*i.e.*, range) of the change, the user keywords, test cases, and test suites that may be affected directly or indirectly by a specific component (or components) can be identified from the coupling diagram. The degree of change impact for the component then can be estimated by calculating the total times the component as well as those affected modules are used. Note that although library keywords directly use components, they are typically offered by KDT tools and, hence, cannot be changed by testers. Thus, the change impact of a component is typically propagated to the corresponding user keywords that perform an action on the component by directly using a library keyword.

When computing the change impact of a component, we take into account the total number of times a module is used. Fig. 8 shows a coupling diagram in which each edge is labeled with a weight showing the number of times a module is used by another module at a higher level. For example, *C*1 is used 10 times by *UK*2, and *UK*2 is used 3 and 10 times by *UK*1 and *TC*1, respectively. Suppose that component *C*4 is changed (*e.g.*, removed, renamed, or replaced by a different component). Then, from Fig. 6, we can observe that the modules directly influenced by *C*4 are *TC*1, *UK*1, and *UK*3. Or, the first-level impact (*i.e.*, an impact length of 1) includes *TC*1, *UK*1, and *UK*3 modules. The second-level (*i.e.*, impact length of 2) includes *TC*1 and *TC*2 that are directly impacted by *UK*1 and *UK*3, respectively. Note that in this case *TC*1 is not only impacted directly at level 1 but also impacted indirectly at level 2 (through the uses of *UK*1). Thus, the impact value at level 1 for *C*4 is 1+2+1=4 and at level 2 is 4+4+4=12, *i.e.*, the impact value at level 1 pluses the values on the edges of *TC*1-*UK*1 and *TC*2-*UK*3. Table 2 shows the estimated impact values for each component at different levels.

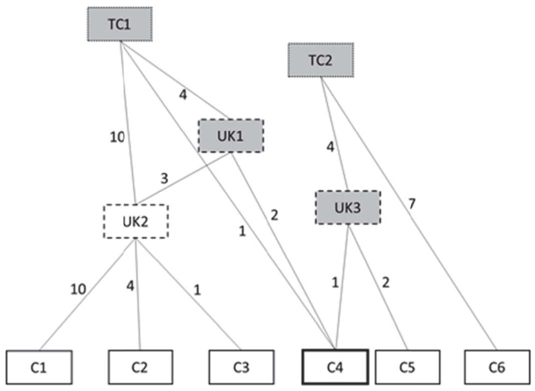


Fig. 8. A coupling diagram with how many times a module has been used.

Table 2. The estimated change impacts at different levels.

Component ID	Level 1	Level 2	Level 3
C1	10	23	27
C2	4	17	21
C3	1	14	18
C4	4	12	12
C5	2	6	6
C6	7	7	7
Total	28	79	91

Eqs. (2) to (5) are the formulas used to calculate the estimated impact values for a component at different levels (*i.e.*, impact length). Let *m* or *n* be a node (representing a user keyword or test case) and *c* be a component in the coupling diagram. The edge between two nodes is denoted as (*m*, *n*) and the edge between a node and a component is denoted as (*n*, *c*). Let *Use*(*x*, *i*) be a set of nodes that use *x* and can be affected by *x* with an impact length of *i* where *x* can be a node or a component. Thus, *Use*(*c*, 1) represents a set of nodes affected by *c* with an impact length of 1. Let *E*(*c*, *k*) be a set of edges between the nodes that can be affected by *c* within an impact length of *k*. Thus, *E*(*c*, 1) represents the set of edges between *n* and *c*, where $n \in Use(c, 1)$, and *E*(*c*, 2) will be the set of edges *e*, where *e* includes *E*(*c*, 1) together with the set of edges (*m*, *n*), where $m \in Use(n, 1)$ and $n \in Use(c, 1)$. Let *CI*(*c*, *k*) be the impact value for component *c* with an impact length of *k*. Thus, *CI*(*c*, *k*) can be calculated using Eq. (5) which is the total sum of *weight*(*e*), where $e \in E(c, k)$ and *weight*(*e*) is the weight of edge *e* representing how many times a module has been used through the edge.

$$E(c,\ 1) = \bigcup_{n \in Use(c,1)} \{(n,\ c)\} \tag{2}$$

$$E(c,\ 2) = E(c,1) \ \bigcup \ (\bigcup_{n \in Use(c,1) \ \wedge \ m \in Use(n,1)} \{(m,n)\}) \tag{3}$$

$$E(c,k) \ = \ (\bigcup_{i=1}^{k-1} E(c,i)) \ \bigcup \ (\bigcup_{n \in Use(c,k-1) \ \wedge \ m \in Use(n,1)} \{(m,n)\}) \tag{4}$$

$$CI(c,\ k) = \sum_{e \in E(c,\ k)} weight(e) \tag{5}$$

The results of a change impact analysis can be shown in a diagram similar to TS-TC-K-C diagram, which will be discussed in the next section. In Section 5, we will further address whether impact values calculated from the above equations reflect real-world change impacts.

4. SYSTEM DESIGN

To support the coupling analysis and visualization of KDT scripts, a supporting tool called KTV (Keyword-driven Test script Visualizer) has been designed and implemented. KTV enhances RIDE [29], a widely used open source integrated development environment for Robot Framework [3]. Fig. 9 shows the system architecture of KTV that consists of five major components, including the Model Builder, Component Recognizer, Coupling Analyzer, Change Impact Analyzer, and Graphviz Generator. The Model Builder is responsible for creating a hierarchical keyword structure model from the input KDT script. The Component Recognizer is used to identify the GUI components associated with the keywords when creating the hierarchical model. The Graphviz Generator converts the model into various diagrams showing the coupling relationships among test cases, keywords, and GUI components using the format of Graphviz [30]. The Coupling Analyzer and the Change Impact Analyzer are in charge of calculating the coupling metrics and possible change impacts based on the model, respectively. The results of coupling metrics and change impact analysis are then displayed using D3.js [31], a popular open source JavaScript library for data visualization.

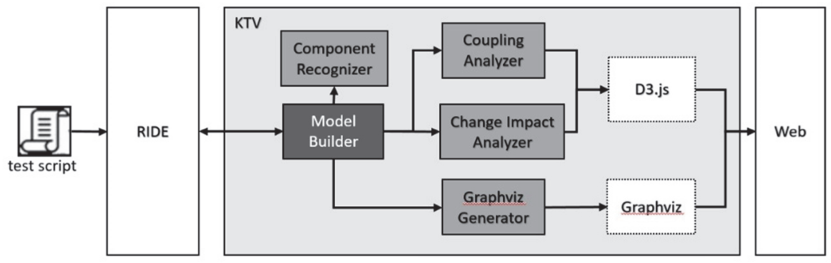


Fig. 9. The system architecture of KTV.

KTV performs both static and dynamic analysis for the generation of diagrams. By analyzing the syntax tree of the target script (*i.e.*, static analysis), the relationships between TS, TC, UK, and LK in Fig. 6 can all be identified. However, identifying the associations between LK and C is more complicated. For example, suppose the library keyword "click okButton" is used. From the syntax tree of the script, we can identify that the target of the click library keyword is okButton. Thus, we know that click is associated with okButton. However, suppose the target button is a variable, *e.g.*, "click ${button}". The value of ${button} cannot be easily identified during static analysis. Thus, dynamic analysis is also required. What KTV does is to request the user to execute the test script for at least once, before generating diagrams. During test execution, KTV Component Recognizer automatically records the target component of every library keyword, creating the associations between LK and C. This information is then used to generate the TS-TC-K-C diagram and TC-UK-C diagram.

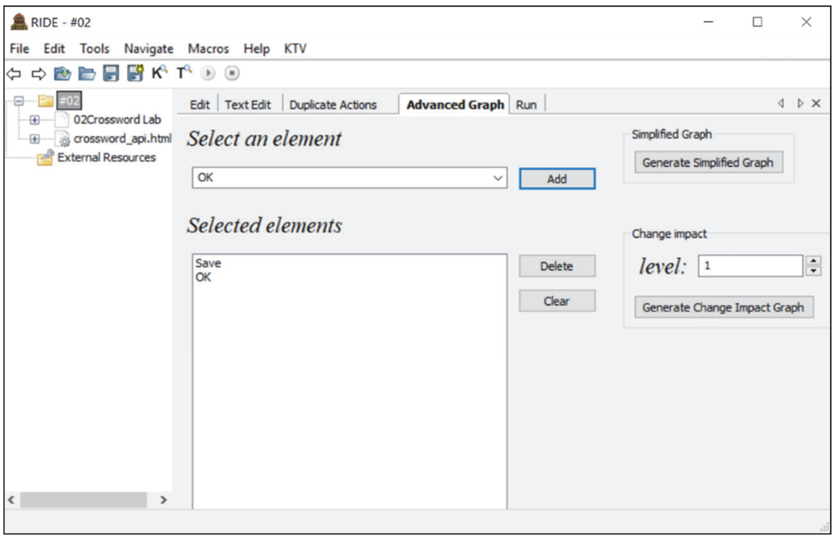


Fig. 10. RIDE user interface enhanced with KTV.

Fig. 10 shows the user interface of RIDE enhanced with the support of KTV. The user uses KTV menu item in the menu to generate and view the eight different diagrams discussed in Section 3.2. Fig. 7 is an example TS-TC-K-C diagram generated by KTV. To improve readability, KTV uses different colors to present different types of modules. To offer a more compact display, a module can optionally be labeled with a short identifier, rather than its full name. In this case, when the mouse hovers on top of a module, KTV shows the full name of the module.

To perform change impact analysis, the user specifies the desired impact level and a list of components that are to be changed by using the user interface (Advanced Graph) shown in Fig. 10, and then KTV automatically generates a change impact diagram like Fig. 11. The diagram shows the modules possibly impacted by the specified changes. In Fig. 11, the component C005 (highlighted using red color) is to be changed and the level of change impact is 2. All the modules within 2 levels from C005 are shown in purple color, including two library keywords (L04 and L06) and five user keywords (U05, U06, U07, U11, and U17). The impact value is 5, which is shown in the rounded rectangle on the left of the root node (S00).

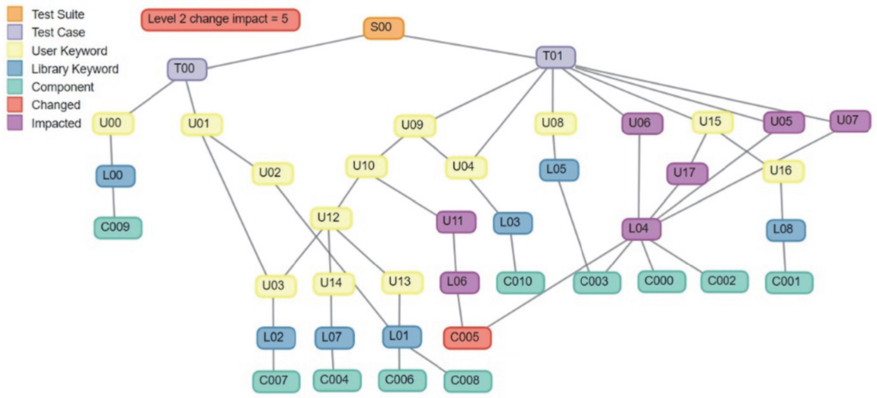


Fig. 11. A change impact estimation example (C005 is the component to be changed).

5. CASE STUDY

This section reports a case study that addresses the following five research questions:

*RQ*1 Given the same test cases (*i.e.*, the same sequence of test actions), do different testers create different keyword designs?

*RQ*2 Does a different keyword design produce a different degree of coupling? What does it look like?

*RQ*3 When maintaining a test script, does its coupling affect maintenance cost?

*RQ*4 When reading a test script, does its coupling affect readability?

*RQ*5 When a component is changed, is the estimated change impact offered by KTV highly correlated with the actual maintenance cost?

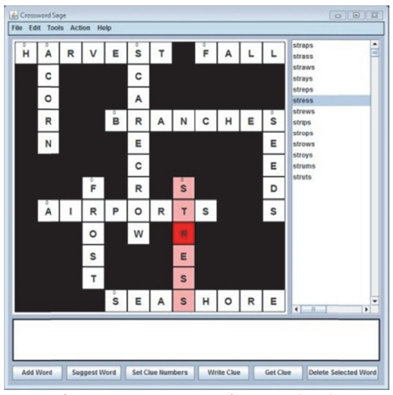


Fig. 12. Crossword Sage (CS).

We conduct four experiments to answer the above research questions. The experiments use Crossword Sage v.0.3.5 (called simply CS) [32] as the target software under test. CS is a rich client application (Fig. 12) that can be used to create/solve crossword puzzles. We choose CS for our experiments because it has been tested by many GUI testing researches [33-36].

5.1 Experiment I

The first experiment addresses *RQ*1 and *RQ*2. We recruit 5 graduate students to participate the experiment. The participants, called P1-P5, act as testers who are requested to develop a KDT script that implements a pre-defined test plan. The test plan contains a total of 9 different test cases, testing the most important user scenarios of CS. Each of the first 7 test cases exercises and tests a single, simple feature. The last 2 test cases, on the other hand, perform integration tests that use several features altogether to create a real crossword puzzle. The test plan specifies the exact sequence of actions that are performed in each test case. A total of 422 actions are required.

Table 3. Coupling of the scripts created by participants P1-P5.

Participant ID	Keywords used	UC
P1	16	1.94
P2	15	1.87
P3	14	2.21
P4	0	2.44
P5	13	1.95

Each participant is requested to use Robot Framework [3] to implement a KDT script that can produce exactly the same 422 test actions. When developing a script, the participants are reminded of keeping maintainability and readability in mind. When a participant completed his script, we perform a verification to ensure that the script is implemented correctly. Then, we use KTV to view the keyword structure and report the coupling of the script. The results are shown in Table 3. It can be seen that every participant designed keywords somewhat differently. For example, P2 used 15 keywords, P3 used only 14, and P4 used no keywords at all. Thus, the answer to *RQ*1 is "yes – while the participants were requested to develop exactly the same test cases (and exactly the same sequence of actions), each participant created a completely different keyword design."

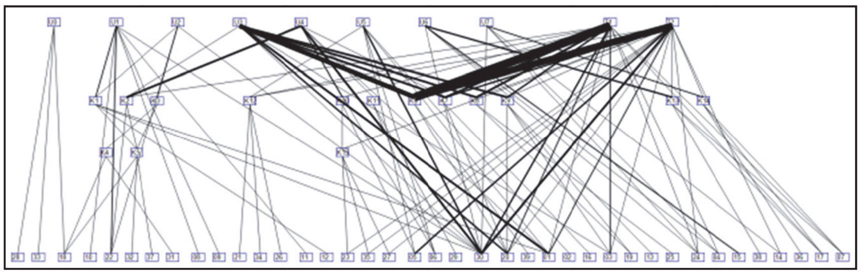


Fig. 13. The coupling diagram of P2 script (note: an edge (K_1, K_2) is drawn with a thicker line when K_1 accesses K_2 for more than once.

Figs. 13 and 14 show the coupling diagrams of P2 script and P3 script, respectively (note: for simplicity, we call P2's script simply P2 script). In P3 script (Fig. 14), many test cases directly access components without using any intermediate keywords. P2 script (Fig. 13), on the other hand, had a better keyword hierarchy. Thus, Fig. 14 shows an overall higher coupling than Fig. 13. The *UC* of P2 script was 1.87, lower than that (2.21) of P3 script. To save space, we do not present all five coupling diagrams. Overall speaking, the answer to *RQ*2 is "yes, a different keyword design did result in a different degree of coupling."

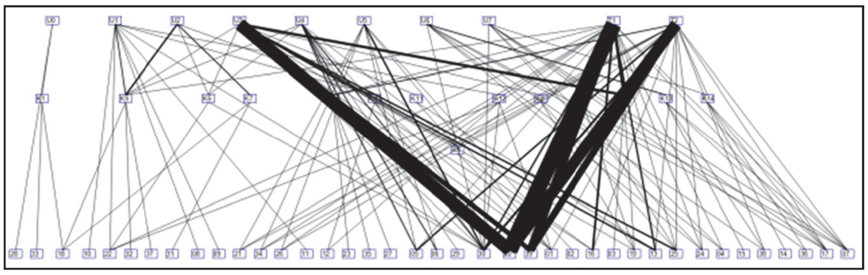


Fig. 14. The coupling diagram of P3 script.

5.2 Experiment II

The second experiment addresses *RQ*3. To evaluate maintainability, we modify the GUI of CS (via modifying its source code). The new GUI results in an improved CS, called CS', which simplifies the following user interactions: (1) remove the "Suggest Word" button and show the suggested words automatically, (2) remove the "Find Possible Matches" button and show possible matches automatically, (3) remove "Add Word" button and use double click instead, and (4) add a new "Add Word Tips" dialog to remind the user of how to add a new word.

A test script designed for CS no longer works for CS'. We request each participant of the first experiment (P1-P5) to maintain (repair) his own script so that the script can be reused to test CS'. The results are shown in Table 4, where "Modifications" indicates the total number of modifications that was made (including both test case and keyword modifications). The correlation coefficient between UC and modifications was 0.97. The strong correlation indicates that a low-coupling script generally required less modifications, *i.e.*, less maintenance cost. Overall speaking, the answer to *RQ*3 is "yes, a low-coupling script generally requires a lower maintenance cost."

Table 4. Test script maintenance cost.

Participant ID	Modifications
P1	45
P2	53
P3	93
P4	140
P5	69

5.3 Experiment III

The third experiment addresses *RQ*4 and then revisits *RQ*3. We recruit another 4 graduate students to participate the experiment. The participants, called P6-P9, are requested to assess the readability of the scripts developed by P1-P5 from experiment I. Each participant (P6-P9) is given all the 5 scripts in random order, and is requested to read each script and then assign a readability grade for each script, based on the grading standard defined in Table 5. Fig. 15 shows the average readability grade received by each script. Except for P2 script, which received a higher grade, the rest of the scripts received similar grades, indicating that these scripts had similar readability. The correlation coefficient between *UC* and readability grade was –0.59. Here, the correlation was not that strong. Our interview with P6-P9 indicated that whether a test script was easy to read depended more on keyword naming than on keyword structures. Some participants pointed out that, when a keyword was not properly named, one needed to constantly re-read its actions so as to confirm its action sequences; some participants considered that P4 script was not any more difficult to read even though it had no keywords at all. This was mainly because the test scripts contained mostly straightforward actions (without any complicated logics), and therefore, the lack of structure did not pose serious readability problems. Thus, the answer to *RQ*4 is a vague "yes and no, there is a correlation between coupling and readability, but the correlation is not very strong – keyword naming is also an important factor to readability."

Table 5. Readability grades.

Grade	Description
5	The test script is very easy to understand. The actions that are to be executed are very clear.
4	The test script is easy to understand. The actions that are to be executed are clear.
3	Most of the test script can be understood. There are occasionally some actions that are not so clear.
2	The test script is difficult to understand. There are a lot of actions that are not so clear.
1	The test script is very difficult to understand. The actions that are to be executed are not clear at all.

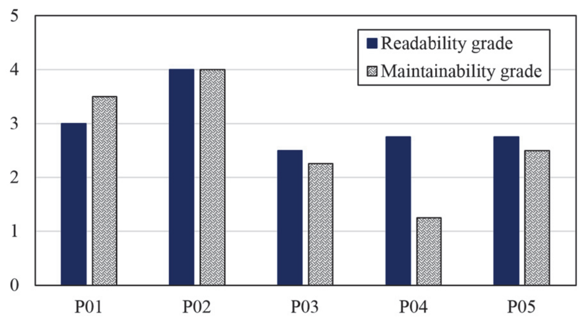


Fig. 15. Average readability and maintainability grade of each script.

We then revisit *RQ*3 by requesting each participant (P6-P9) to assess the maintainability of P1-P5 scripts. We inform the participants that the GUI is to be changed from CS to CS' (the same changes as described in Section 5.2). Each participant is requested to identify how each of the five scripts can be repaired and reused for CS'. Based on the definition given in Table 6, each participant assigns a maintainability grade for each script. The results are shown in Fig. 15. Note that, while P4 script was not particularly bad in readability, it received the worst maintainability grade, indicating that a test script could be easy to read, yet difficult to maintain. The correlation coefficient between *UC* and maintainability grade was −0.91, a very strong correlation. In other words, from the viewpoints of P6-P9, a script's coupling was highly related to its maintainability. The results reaffirmed the answer to *RQ*3 reported in the previous section.

Table 6. Maintainability grades.

Grade	Description
5	The test script is very easy to maintain. A small GUI change results in only very few keywords (or test cases) need to be changed, and a keyword change does not impact the other keywords (or test cases).
4	The test script is easy to maintain. A small GUI change results in only a small number of keywords (or test cases) need to be changed, and a keyword change does not impact the other keywords (or test cases).
3	The test script is not so easy to maintain. A small GUI change results in many keywords (or test cases) need to be changed, and a keyword change could impact some other keywords (or test cases).
2	The test script is difficult to maintain. A small GUI change results in a large number of keywords (or test cases) need to be changed, and a keyword change could impact many other keywords (or test cases).
1	The test script is very difficult to maintain. A small GUI change results in almost all keywords (or test cases) need to be changed, and a keyword change could impact almost all other keywords (or test cases).

5.4 Experiment IV

The fourth experiment addresses *RQ*5. As described in the previous experiment, when the GUI is changed, some script modifications are unavoidable. But, before the change, could we estimate the scale of the change so that we can be more confident about the change? KTV offers an estimation of change impact based on the coupling between component and keywords (Section 3.3). This experiment studies whether such an estimated change impact is realistic.

What we would like to do in this experiment is to estimate the change impact of changing from CS to CS'. For this purpose, we assume that we do not have CS' yet. First, we create a change list that contains all the components to be changed in CS (the changes described in Section 5.2). For example, the change list contains "Suggest Word" button, because it is going to be removed. Note that the fourth GUI change adds a brand-new component ("Add Word Tips" dialog) to CS. That is, this new component is not a part of CS and certainly cannot be a member of the change list. Thus, we use a component in CS that is closely related to the add word operation and put it into the change list to stand for the new component.

We then use KTV to estimate the change impact. For each script (P1 script – P5 script), we enter the above change list into KTV (through KTV user interface). KTV then calculates the change impact based on these components. The results for level-1 impact values are shown in Fig. 16. For example, the estimated change impact of P2 script was

29, indicating that once the components in the change list are changed, 29 different places (test cases/keywords) that directly use these components are affected and need to be modified. This is a lower bound of the changes that are required. From Fig. 16, the correlation coefficient between the actual maintenance cost (the modifications shown in Table 4) and the estimated change impact was 0.99, indicating that the estimated change impact could be used as a reliable indicator for estimating the actual maintenance cost. Although, the ratio between actual maintenance cost and estimated change impact is unknown beforehand, such ratio could be obtained from history records. For example, suppose the tester keeps tracks of some previous maintenance records, including maintenance cost (efforts) and estimated change impact values. This history information can then be used to project the maintenance cost of a future change.

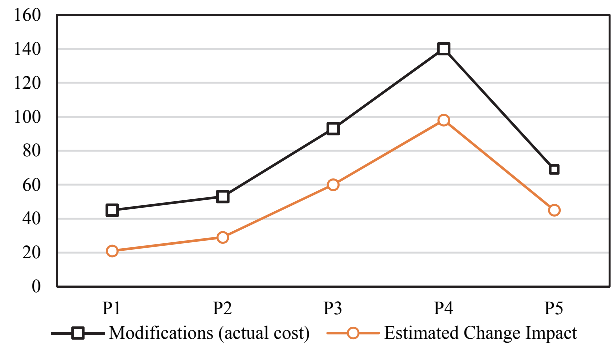


Fig. 16. The estimated level-1 change impact versus the actual maintenance cost.

We also use KTV to calculate the impact values for different levels. The results indicated that the impact values beyond level 1 (*e.g.*, level 2) were significantly over estimated. This was because, when a component was changed, user keywords provided a mechanism of encapsulating the change and thus the change impact did not propagate to the next level. Overall speaking, the answer to *RQ*5 is "yes, the estimated change impact offered by KTV is highly correlated with the actual maintenance cost."

6. CONCLUSIONS AND FUTURE WORK

This paper studies the analysis and visualization of KDT scripts. A measure based on the density of coupling diagram has been proposed. Eight different diagrams are presented and allow testers to visualize the coupling in a hierarchical way so as to quickly grasp the structure of a script. In addition, a change impact estimation of KDT scripts based on the ripple effect of the coupling dependencies is described. We designed and implemented a coupling analysis and visualization tool called KTV. A case study with four experiments is conducted to assess the relationships between coupling and maintainability as well as whether change impact estimation is useful. The results indicated that, when maintaining a test script, a low-coupling script required, on average, less maintenance cost. Moreover, the change impact estimation was realistic and could be valuable for assessing the maintenance cost of KDT scripts. In addition, keyword naming is also an important factor to readability. Thus, for a tester who needs to constantly develop/maintain KDT scripts, it is advisable that the test script is designed with both low-coupling and good keyword naming.

In the future, we plan to extend the coupling measure to take into account the number of times a module is used. Moreover, the hierarchical diagrams shown by KTV currently support only limited user interactions. Thus, we also plan to extend KTV so that the diagrams can support more user interactions to facilitate coupling analysis and visualization of KDT scripts.

ACKNOWLEDGMENT

This research was partially supported by the Ministry of Science and Technology, Taiwan, under contract numbers MOST 105-2221-E-027-085 and MOST 105-2221-E-027-086, which is gratefully acknowledged.

REFERENCES

1. W. K. Chen, Z. W. Shen, and T. H. Tsai, "Integration of specification-based and cr-based approaches for GUI testing," *Journal of Information and Science Engineering*, Vol. 24, 2008, pp. 1293-1307.
2. A. M. Memon, M. E. Pollack, and M. L. Soffa, " Hierarchical GUI test case generation using automated planning," *IEEE Transactions on Software Engineering*, Vol. 27, 2001, pp. 144-155.
3. Robot framework, http://robotframework.org/.
4. M. Fewster and D. Graham, *Software Test Automation*, Addison-Wesley Professional, London, 1999.
5. HP: Unified functional testing (UFT), http://en.wikipedia.org/wiki/HP_QuickTest_Professional.
6. C. Kaner, J. Bach, and B. Pettichord, *Lessons Learned in Software Testing: A Context-Driven Approach*, Wiley, NJ, 2001.
7. W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Systems Journal*, Vol. 13, 1974, pp. 115-139.
8. G. J. Myers, *Reliable Software through Composite Design*, Petrocelli/Charter, NY, 1975.
9. L. L. Constantine and E. Yourdon, *Structured Design*, Prentice-Hall, NJ, 1979.
10. C. Larman, *Applying UML and Patterns*, 3rd ed., Prentice Hall PTR, NJ, 2004.
11. D. A. Troy and S. H. Zweben, "Measuring the quality of structured designs," *Journal of Systems and Software*, Vol. 2, 1981, pp. 113-120.
12. N. Fenton and A. Melton, "Deriving structurally based software measures," *Journal of Systems and Software*, Vol. 12, 1990, pp. 177-187.
13. H. Dhama, "Quantitative models of cohesion and coupling in software," *Journal of Systems and Software*, Vol. 29, 1995, pp. 65-74.
14. D. Kafura and S. Henry, "Software quality metrics based on interconnectivity," *Journal of Systems and Software*, Vol. 2, 1981, pp. 121-131.

15. S. Henry and D. Kafura, "Software structure metrics based on information flow," *IEEE Transactions on Software Engineering*, Vol. SE-7, 1981, pp. 510-518.
16. A. J. Offutt, M. J. Harrold, and P. Kolte, "A software metric system for module coupling," *Journal of Systems and Software*, Vol. 20, 1993, pp. 295-308.
17. P. Coad and E. Yourdon, *Object-Oriented Analysis*, 2nd ed., Prentice Hall, NJ, 1991.
18. S. R. Chidamber and C. F. Kemerer, "Towards a metrics suite for object oriented design," in *Proceedings of Conference on Object-Oriented Programming Systems*, *Languages*, *and Applications*, 1991, pp. 197-211.
19. S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, Vol. 20, 1994, pp. 476-493.
20. J. Eder, G. Kappel, and M. Schrefl, "Coupling and cohesion in object-oriented systems," Technical Report, University of Klagenfurt, 1994.
21. M. Hitz and B. Montazeri, "Measuring coupling and cohesion in object-oriented systems," in *Proceedings of International Symposium on Applied Corporate Computing*, 1995, pp. 25-27.
22. L. C. Briand, J. W. Daly, and J. K. Wüst, "A unified framework for coupling measurement in object-oriented systems," *IEEE Transactions on Software Engineering*, Vol. 25, 1999, pp. 91-121.
23. J. Offutt, A. Abdurazik, and S. R. Schach, "Quantitatively measuring object-oriented couplings," *Software Quality Journal*, Vol. 16, 2008, pp. 489-512.
24. R. W. Selby and V. R. Basili, "Analyzing error-prone system structure," *IEEE Transactions on Software Engineering*, Vol. 17, 1991, pp. 141-152.
25. K. E. Emam, W. Melo, and J. C. Machado, "The prediction of faulty classes using object-oriented design metrics," *Journal of Systems and Software*, Vol. 56, 2001, pp. 63-75.
26. F. G. Wilkie and B. A. Kitchenham, "Coupling measures and change ripples in C++ application software," *Journal of Systems and Software*, Vol. 52, 2000, pp. 157-164.
27. Z. Jin and A. J. Offutt, "Coupling-based criteria for integration testing," *Software Testing*, *Verification*, *and Reliability*, Vol. 8, 1998, pp. 133-154.
28. W. K. Chen, C. H. Liu, P. H. Chen, and Y. Wang, "Is low coupling an important design principle to KDT scripts?" in *Proceedings of International Conference on Frontier Computing*, Vol. 422, 2016, pp. 45-56.
29. RIDE, http://www.github.com/robotframework/RIDE/.
30. Graphviz, http://www.graphviz.org.
31. D3.js, http://d3js.org.
32. B. Westgarth, "Crossword sage," http://crosswordsage.sourceforge.net.
33. W. K. Chen and J. C. Wang, "Bad smells and refactoring methods for GUI test scripts," in *Proceedings of the 13th International Conference on Software Engineering*, *Artificial Intelligence*, *Networking and Parallel Distributed Computing*, 2012, pp. 289-294.
34. A. M. Memon, "Automatically repairing event sequence-based GUI test suites for regression testing," *ACM Transactions on Software Engineering and Methodology*, Vol. 18, 2008.
35. Q. Xie and A. M. Memon, "Using a pilot study to derive a GUI model for automated testing," *ACM Transactions on Software Engineering and Methodology*, Vol. 18, 2008.

36. X. Yuan and A. M. Memon, "Generating event sequence-based test cases using GUI runtime state feedback," *IEEE Transactions on Software Engineering*, Vol. 36, 2010, pp. 81-95.

Chien-Hung Liu (劉建宏) received his Ph.D. degree in Computer Science and Engineering from the University of Texas at Arlington in 2002. He is currently an Assistant Professor of Computer Science and Information Engineering Department at National Taipei University of Technology, Taiwan. His research interests include software testing, software engineering, and cloud computing.

Woei-Kae Chen (陳偉凱) received M.S. and Ph.D. degrees in Computer Engineering from North Carolina State University in 1988 and 1991, respectively. He is currently a Professor at the Department of Computer Science and Information Engineering and the director of Software Development Research Center of the National Taipei University of Technology, Taiwan. His research interests include software testing, software engineering, visual programming, and cloud computing.