Journal of Theoretical and Applied Information Technology
20th February 2016. Vol.84. No.2

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

A META-MODEL FOR AUTOMATIC MODELING DYNAMIC WEB APPLICATIONS

TANYA SATTAYA-APHITAN[1], HORST LICHTER[2], TONI ANWAR[3], SANSIRI TANACHUTIWAT[4]

[1]Software System Engineering (SSE), The Sirindhorn International Thai-German Graduate School (TGGS), KMUTNB, Bangkok, Thailand

[2]Research Group Software Construction, RWTH Aachen University, Ahornstr. 55, Aachen, Germany

[3]Faculty of Computing (FC), Universiti Teknologi Malaysia (UTM), 81310 Johor Bahru, Malaysia

[4]Software System Engineering (SSE), The Sirindhorn International Thai-German Graduate School (TGGS), KMUTNB, Bangkok, Thailand

E-mail: [1]tanya.s-sse2014@tggs-bangkok.org, [2]lichter@swc.rwth-aachen.de, [3]tonianwar@utm.my, [4]sansiri.t.sse@tggs-bangkok.org

ABSTRACT

This paper proposes an approach to automatically transform source code of a web application into an abstraction model. A Web Application Program Dependency (WAPD) meta-model is being proposed to store dependency information based on a multi-tiered architecture, corresponding to web application's behavior. A WebParseTree is used as an intermediate model for the transformation from the source code to the WAPD model. The WebParseTree is a DOM-like tree that consists of statements and dependencies stored information and behavior in the tree. To ensure that the resulting model is valid, it must conform to the defined web application rules. This validation step can be done automatically by a constraint validator using Object Constraint Language (OCL). The WAPD model will be represented as a generic model for web applications which can be used for many purposes such as automatic test case generation and automatic code transformation.

Keywords: *Web Parse Tree, Web Application Modeling, Web Application Meta-model, Data Object Modeling (DOM), Web Application Automatic Transformation.*

1. INTRODUCTION

Nowadays many organizations are increasingly using web applications for e-business/e-commerce. Hence, it is important to ensure the required quality of web applications before deploying them because one failure could result in significant losses. One of the essential methods to assure the quality is to systematically test an application. Two fundamental techniques to determine a set of test cases are functional and structural testing, also known as black-box and white-box testing. These testing techniques concern two different perspectives. Black-box tests software are based only on the specification while white-box tests are based on the internal structure and the specification of the application under test. Structural and functional testing are complementary. Web application testing tools (e.g. Selenium, HTMLunit, JWebUnit) while supporting functional testing, do not offer structural testing[1], and are therefore incomplete.

This paper presents an approach to automatically transform the source code of a web application into an abstraction model that can be used to systematically derive test cases. However, creating an abstraction model of a browser based web application is much more complicated compared to desktop applications due to its multi-tiered or client-server architecture.

Normally, a web application is composed of three tiers as shown in Figure 1. Tier 1 (client tier) is interacting with end users while tier 2 (server tier) is processing the business logic. Tier 3 (data tier) is performing database transactions or communicates to other web applications via web service requests.

Journal of Theoretical and Applied Information Technology
20th February 2016. Vol.84. No.2

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

For the client tier, input data validations or calculations should be done by means of client-side scripts (e.g. JavaScript, VBScript). A web browser is used as a client to host the web application and renders its client-side components, such as HTML, client-side scripts, applets, that interact with the users.

On the server tier, the business logic is often realized by means of server-side components implemented in various programming languages such as PHP, ASP, JSP, Java and VB. After receiving and processing HTTP request the server sends HTTP responses back to the client which displays the result. Thus, structural testing of web applications has to deal with analyzing the program execution paths on both client and server tier implemented in different programming languages.

Moreover, both tiers are spatially separated and communicate with each other using the HTTP protocol, a stateless protocol, meaning that one must take special care of handling the transmission of parameters among them. The above mentioned limitations pose great challenge to transform web applications to a model.

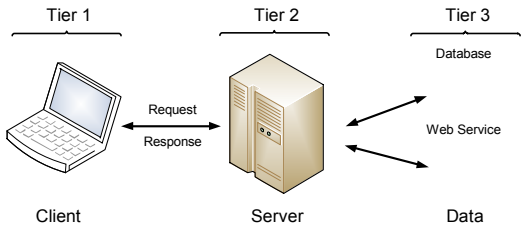

Figure 1: Structure of multi-tiered web applications.

The remainder of this paper is organized as follows: section 2 reviews existing work in web application testing. Section 3 proposes an approach to model web applications. A meta-model is presented to represent intermediate information. The implementation details are introduced in section 4. The preliminary result and conclusions are summarized in section 5.

2. RELATED WORK

This section briefly surveys related works on model-based test of web applications. The related work can be classified as follows:

2.1 Static Webpage Modeling

Ricca and Tonella [2] model web applications using UML. They proposed the tools called ReWeb and TestWeb. ReWeb collects static web pages from the website and represents them as a UML model. Then test cases are generated by TestWeb. However, they consider only static web pages without considering dynamic ones. Reza et al. [3] applied state charts to model web applications comparing three different kinds: FSMs , Petri nets and state charts. However, they mentioned that "we have not yet found solutions to the problems of modeling concurrency and modeling the back-ends of web applications". This work also considers only static web pages. Rafique et al. [4] model web applications using FSM. The model is represented as a graph where nodes represent the pages and edges represent the page navigations. The FSM transformation is done manually and only on page level. Likewise, Machra and Khatri [5] use directed graph to model web applications. Nodes represent pages while edges represent hyperlinks. Both graph models in [4] and [5] were done on page level and did not consider client and client/server-side scripting.

2.2 Server-Side Script Modeling

Youxin et al. [6] proposed a test generation framework based on Z specification. PDG is used to model web applications. But, they only introduced the basic idea and has not provided an implemented. Moreover, their approach considers only server-side scripts. Wassermann et al. [7] proposed algorithms for analyzing server-side scripts and for discovering the input data based on the concolic testing approach. This work focuses only on server-side scripts.

2.3 Client-Side Script Modeling

Artzi et al. [8] proposed a technique for generating concrete input data based on feedback-direct random testing. The technique focuses on testing java script, yielding an average coverage of 69%. Mesbah et al. [9] introduced a methodology for testing AJAX applications by crawling in a state flow graph on the client-side. This approach can automatically detect faults by comparing the state change with the DOM-tree serving as an oracle.

2.4 Multi-Tier Modeling

In regard to the actual architecture of web applications, it is not enough to model only one of the tiers. There are many research studies on modeling multi-tier web applications. Dia et al. [10] proposed a methodology for modeling multi-tier web applications. The client tier is modeled by means of WGUI trees. The server tier is modeled as a system dependency graph, and the data tier is modeled as a data object tree. They proposed INSDG as a model to integrate the tier. The concrete input data is generated by using a symbolic execution technique together with a

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

boundary value analysis. However, they did not consider client-side scripts in their approach. Moreover, their approach requires a use case based specification. Ricca and Tonella [11] proposed an approach to model web applications with two layers: the navigation model and the control flow model whose coverage metric is calculated in each layer separately based on code instrumentation. Test cases can be generated from the model. However, this approach somehow required a human-assistant to create the model. Gu et al. [12] introduced the approach to model three web application components which consist of web server, application server and database server. The control dependencies on each component are built and connected together with message dependencies. However, this approach lacks a methodology for test case generation. Tung et al. [13] proposed a novel approach to model web applications. It consists of two phases which are the test path analysis phase and test case generation phase. In phase 1, a path navigation diagram is created based on data and control dependencies. The proposed algorithm eliminates cycles from a path navigation diagram to yield a primitive path and simple cycle which is used as test path. In phase 2, a test case generation algorithm is applied to the primitive path by considering input values and the dependencies. The input values have to be defined manually. However, this approach lacks input data generation. It is done on the page level and does not provide coverage metrics and expected results. Sabharwal et al. [14] proposed a Page Navigation Graph (PNG) to model web applications. The PNG is created from information on low level design (DTD) containing page and window scenarios. This work focuses only on page/windows level. Bansal and Sabharwal [15] proposed a method to convert a PNG [14] to a Control Flow Graph (CFG). The CFG is then traversed to generate test case sequences. Achkar [16] proposed a FSM to model the navigation behavior of web applications by means of its states and the action change related to its state. He applied a FSM model with TestOptimal framework to generate test cases. Carcia and Duenas [17] proposed an automated page navigation modeling technique by means of UML diagrams, Record and PlayBack (R&P) XML. These were treated as inputs to a tool, called Automated Testing Platform (ATP), to create multi-digraph. The Chinese Postman Problem (CPP) was used to generate test sequence from the multi-digraph. This method provides support to generate test data and test oracle.

Most of the related works propose to model web applications on the page level. This paper considers to *automatically* modeling web applications on the *source code level*. The approach analyzes both the *client and the server pages*. In addition, the resulting model integrates the client and the server part in one single model. This *model* can be applied to generate test cases or to transform the source code. As the proposed model is a white-box model its internal structure can be analyzed e.g. to *measure* the code coverage.

3. PROPOSED APPROACH

The following section introduces the proposed approach to automatically transform source code to an abstraction model. The Web Application Program Dependency (WAPD) meta-model represents the source code, its structure and dependencies. A Code-to-Model Transformation (C2M) is introduced to transform the web application's source code to a WAPD model conforming to the WAPD meta-model as shown in Figure 2.

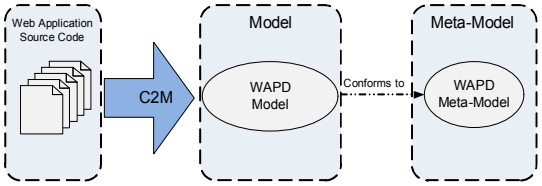

Figure 2: Proposed Web Application Modeling Approach.

3.1 Code to Model Transformation (C2M)

The C2M-transformation, as shown in Figure 3 can be divided in two steps. First, a Web Application (WA) Parser parses the source code and creates its corresponding WebParseTree (the DOM tree). Second, the WA generator, based on the resulting DOM tree, generates accordingly a WAPD model. This model conforms to the WAPD meta-model which defines all necessary information for generating test cases. In addition, a constraint validator, which is a part of WAPD meta-model, is used while generating the model. The constraints define connection's rules between nodes and dependencies within the proposed graph to produce a proper WAPD model.

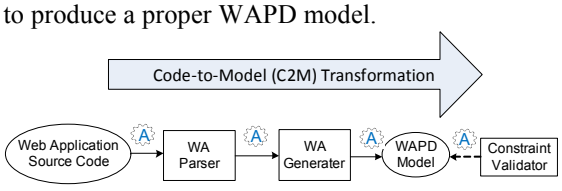

Figure 3: Transformation Of Source Code To WAPD Model.

Journal of Theoretical and Applied Information Technology
20th February 2016. Vol.84. No.2

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

3.2 Web Application Program Dependency Meta-Model (WAPD)

The WAPD meta-model defines a language to represent the source code and its dependencies of a web application. It is adapted from PDG [18] to accommodate the diversity of web programming languages such as HTML, client-side scripting, server-side scripting. Normally, a PDG contains only two kinds of dependency: control and data dependency. The meta-model will be enhanced by event dependencies to represent the web application's behavior.

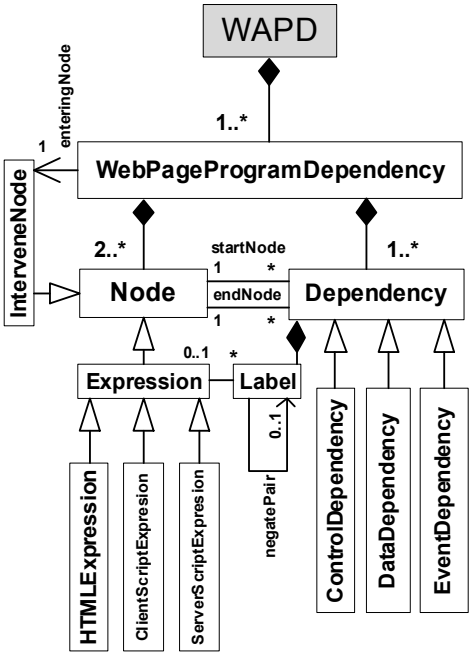

Figure 4: The WAPD Meta-Model.

Based on the WAPD meta-model, shown in Figure 4, a WAPD model can consist of many web pages. A web page is represented as a graph structure called $WebPageProgramDependency$ (WPPD). It consists of *Nodes* and *Dependencies*. A $Node$ can be either an $Expression$ or an $InterveneNode(N_I)$ depending on the information contained in the node.

An $Expression$ can be differentiated into three types:

- $HTMLExpression(E_H)$,
- $ClientScriptExpression(E_C)$, and
- $ServerScriptExpression(E_S)$.

These three node types are used to represent HTML, client-side scripting and server-side scripting respectively. Likewise, a N_I node usually stores irrelevant information. This node is used to represent source code in the case of CSS, applets or embedded objects of web pages.

Furthermore N_I nodes are used for grouping sets of expressions, i.e., E_H, E_C, E_S. This is useful if one wants to model a large web page by separating it into several parts. Moreover, a N_I node is defined as a root node of every web page.

A relationship between *Nodes* is called *Dependency*. There are three different dependency types:

- $ControlDependency$ (D_C),
- $DataDependency$ (D_D) , and
- $EventDependency$ (D_E).

A D_C is used to model the program's execution flow, also called control flow [19]. While a D_D serves to identify a data flow [20] a D_E is introduced to represent event behavior resulting from user interactions.

A $Label(L)$ stores a Boolean expression and is used as a guard associated to D_C and D_E dependencies. The value $true$ means that the program's execution can flow from a source node to a destination node. If the label is evaluated to $false$, no control flow is allowed from the source to the destination node. Normally, every D_C must have a L. If L is not initialize on a D_C, this implies that the value of L_B is $true$ by default. Additionally, every D_E has to be labeled by an intended event for specifying the flow control if the event is handled. A label is not associate with a D_D because data can always be referred at any point of the program.

3.3 Constraint validator

In order to store information in the model, it is necessary to follow the model's constraints, called invariants. The invariants are derived from the actual behavior of web application to prevent an invalid link (or dependency) between each node. The WAPD consists of three invariants depending on each dependency type: D_E, D_D and D_C.

Moreover, each invariant has two types: (1) intra-invariant: the invariant ensures the correctness of constructing a WPPD model within a web page and (2) inter-invariant: the invariant ensures the correctness of a relationship between WPPD stored in a WAPD. This consists of many web pages communicating to each other via request methods.

Figure 5 summarizes the constraint validation rules defined on each dependency type regarding each expression type. There are three dependency constraints:

(1) Constraints on event dependency: A D_E dependency is only allowed to link from E_H to

Journal of Theoretical and Applied Information Technology
20th February 2016. Vol.84. No.2

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

either E_C inside a single page or to N_I between pages.

(2) Constraints on data dependency: A D_D, is only allowed from E_H to E_C and vice versa on a single web page while a D_D is allowed to link from either E_H or E_C to E_S between different web pages.

(3) Constraints on control dependency: Inside a single web page a D_C is allowed between all types of expressions. For the dependency between web pages, all types of expressions (i.e., E_H , E_C or E_S) are allowed to connect to N_I. This process is called a request-response process.

We uses the Object Constraint Language (OCL) [21] to express these invariants and integrated the implemented constraints to the WAPD meta-model. Hence, the constraints are automatically validated on WAPD model. In case constraints are violated, errors will be raised.

Source \ Target		Intra − inv.				Inter − inv.			
		E_H	E_C	E_S	N_I	E_H	E_C	E_S	N_I
D_E	E_H	○	●	○	○	○	○	○	●
	E_C	○	○	○	○	○	○	○	○
	E_S	○	○	○	○	○	○	○	○
D_D	E_H	○	●	○	○	○	○	●	○
	E_C	●	○	○	○	○	○	●	○
	E_S	○	○	●	○	○	○	○	○
D_C	E_H	●	●	●	●	○	○	○	●
	E_C	●	●	●	●	○	○	○	●
	E_S	●	●	●	●	○	○	○	●

● = Allow ○ = Not allow

Figure 5. Dependencies' Constraints On Expressions.

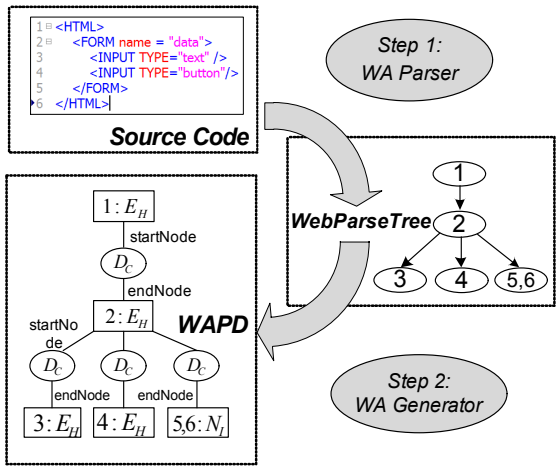

Figure 6. Schematic Transformation Process And Its Artifacts.

4. IMPLEMENTATION DETAIL

In the next section, details of the transformation approach introduced in the previous section are presented which provide some insight into its implementation. A simplified login application shown in Figure 7 has been chosen as an example to demonstrate the transformation process and the intermediate and final results.

As shown in Figure 6, the modeling process involves two steps:

(1) The WA Parser parses the source code and creates the corresponding web application parse tree (WebParseTree), and

(2) the WA Generator transforms the WebParseTree to a respective WAPD model.

4.1 Web Application Parser

At first, the Web Application Parser parses the web application's source code and creates a corresponding web application parse tree based on the following *RegExp* rules [22]:

WA : [WP]+

WP : [PP]+

PP : [HTML]* [CS]* [SS]* [PT]*

Here, WA represents a web application that consists of one or more web pages (WP). A WP may contain one or more Web Portion (PP). A PP may include Plain Text (PT) and three major programming parts, i.e., HTML, client-side scripting (CS) and server-side scripting (SS). If a web page consists only of web portions implemented in HTML, it is called a static web page. Otherwise it is a dynamic web page.

In our example page *login.html* shown in Figure 7 (a), the section starting from line number 1 to 4 is plain HTML and CS (i.e. JavaScript opening tag). It is identified as a PP and labelled P1. The CS section from line number 5 to 13 is the second PP labelled P2. P4 consists of the HTML form starting at line number 17 to 21. The remaining lines are grouped into two more PPs, P3 and P5, covering HTML code from line 14 to 16 and from line 22 to 23 respectively.

```
P1  1  <html>
    2   <head>
    3   <title></title>
    4   <script>
P2  5     function chk(){
    6         var user = document.flogin.user.value;
    7         var pass = document.flogin.password.value;
    8         if(user.length == 0 || pass.length==0 ) {
    9             alert("Please input username/password: ");
   10             return false;
   11         }else return true;
   12     }
P3 13   </script>
   14   </head>
   15   <body>
   16   Welcome to login page.
P4 17   <form name="flogin" action="LoginAction.php" onsubmit="return chk()">
   18     User:<input type="text" name="user"></br>
   19     Password:<input type="text" name="password"></br>
   20     <input type="submit" value="login" name="fsubmit">
   21   </form>
P5 22   </body>
   23 </html>
```

(a) Login.html

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

```
P1  1  <?php
    2  $user = $_GET["user"];
    3  $password = $_GET["password"];
    4  echo("<html><head><title>Welcome</title></head><body>");
    5
P2  6  if(validateUser($user,$password)){
    7    echo("<form name=form action=UserPage.php>");
    8    echo("Please enter your verification code:");
    9    echo("<input name='code' >");
   10    echo("<input type='submit'></form>");
   11  }else{
   12    echo("Invalid user/password!!!");
   13    echo("<A href='login.html'>Try again</A>");
   14  }
P3 15  echo("<A href='index.html'>Main</A>");
   16  echo("</body></html>");
   17
P4 18  function validateUser($user,$password){
   19      if($user =="user")
   20          return true;
   21      else return false;
   22  }
   23  ?>
```

(b) *LoginAction.php*

Figure 7 Code of a simplified login application.

The script contained in P2 has a simple control flow, a sequence of statements (lines 6, 7) and a conditional branch in line 8. A parse tree may be linked with another WebParseTree, e.g. in line 17the control flow requests to proceed at page $LoginAction.php$. The resulting parse tree of our simplified login application is shown in Figure 8. Nodes represent either PPs or line numbers of the source code. Edges model the control flow of the program. If we traverse the tree starting at its root node and applying a depth first search, we get the original source code.

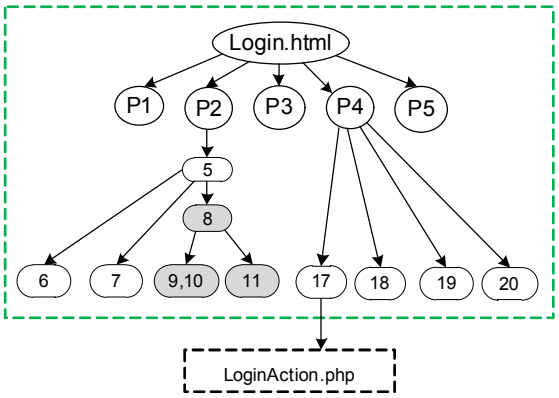

Figure 8: A Webparsetree Of A Login Application Shown In Figure 7.

Generally, a PP may consist of cascaded style sheets (CSS) or embedded web objects. These sections will be modelled as a plain text (PT). The CSS is not considered because it serves the purpose of decorating the web application only. In addition, embedded web objects (e.g., java applet, adobe flash) are also not considered as they sometimes come under third-party libraries.

4.2 Web Application Generator

In this step a WAPD model, as introduced in section 3.2, is created based on the resulting parse tree. A WAPD is a model that keeps all necessary information of a web application to generate white-box test cases. The expression nodes (HTML-, ClientScript-, and ServerScript-Expression – abbreviated to E_H, E_C, E_S respectively) and their dependencies can be constructed based on information contained in the parse tree. According to the presented WAPD meta-model three types of dependency (Control-, Event-, and Data-Dependency – abbreviated to D_C, D_E, D_D respectively) are offered to connect expression nodes together based on their behaviors. The following steps describe the process to build the WAPD model.

4.2.1 Create control dependencies

As mentioned before, the created parse tree itself represents the control flow of a program. If we traverse the tree applying pre-order depth-first search (DFS), we obtain a traditional Control Flow Graph (CFG) [23]. Hence, we can create the control dependencies by a direct mapping from the parse tree to the WAPD model. There are two cases of modeling a control dependency:

(1) Control Dependency without Label

Basically, every control dependency (D_C) is labelled with $'true'(T)$, which means that the control flows immediately from the source to the destination node of the dependency. This T label can be omitted by default as it is a traditional control flow.

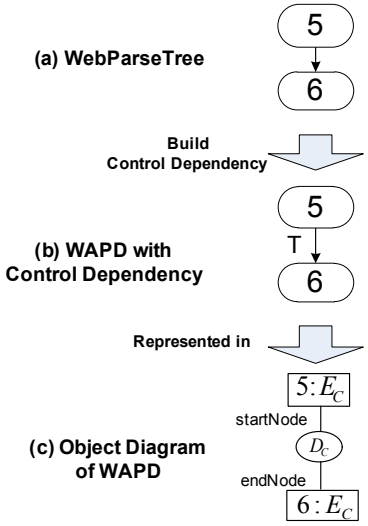

Figure 9. Example Of Building Control Dependency Without Label.

Figure 9(a) shows the simple control flow from node (5) to node (6). This control flow is converted to s D_C dependency and is labelled with T. The transformed WAPD is shown in Figure 9(b). A corresponding object diagram of the WADP model as shown in Figure 9(c) does not include a label

Journal of Theoretical and Applied Information Technology
20th February 2016. Vol.84. No.2

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

object associated with *the* D_C object as the label value T is defined by default. This case is called "*Control Dependency without Label*". Please note that $x{:}\,E_C$ means $ClientScriptExpression$ of node x and $5{:}\,E_C$ means $ClientScriptExpression$ of node 5.

(2) Control Dependency with Label

If we model a conditional flow the associated label holds the respective Boolean condition, which must be evaluated. In case its value is $true$, the control flows from the source node to the destination node of the dependency. In contrast, if its value is *false*, there is no control flow from source node to destination node.

A simple example taking from node (8) to (11) in Figure 8 (if-then-else control flow) is given in Figure 10(a). Figure 10(b) shows the control dependencies of the WAPD model. The condition $A = (\text{user.length} == 0 \,||\, \text{pass.length} == 0)$ is associated with the control dependency from node (8) to node (9,10), while its negation $!A$ is associated with the control dependency from node (8) to node (11).

A control dependency (D_C) links a client-side script on node 8 ($8{:}\,E_C$) to a client-side script on node 9,10 ($9{,}10{:}\,E_C$). This is labelled with A (L_A). On the other hand, node 8 ($8{:}\,E_C$) is linked to node 11 ($11{:}\,E_C$) by a control dependency that is labelled with $A!$ ($L_{A!}$). Figure 10(c) shows a corresponding object diagram of the WADP model.

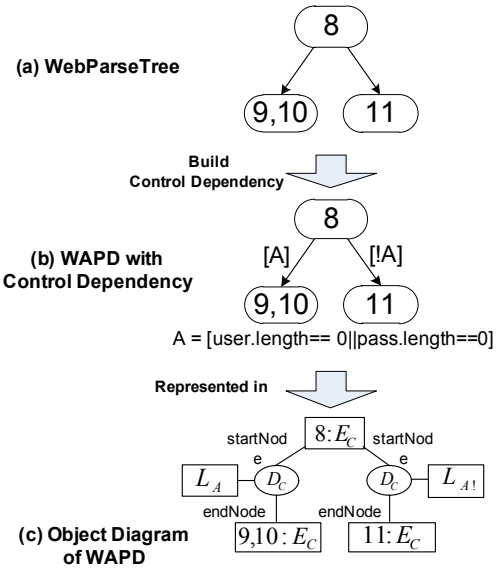

Figure 10: Example of building Control Dependency with Label.

Figure 11 presents the recursive algorithm to create the control dependencies. As initial input the root node (v) of the parse tree is passed to the algorithm, the parameter wapd is null. At first, v is marked as a visited node. Then, a WAPD is created and v is defined to be the entering node of the WAPD (lines 8 and 9). After that, the algorithm is called recursively on every child node (w) of v (line 13). If w is not marked as visited, the WAPD is modified by adding a control dependency from v to w (line 15), followed by a recursive call on w (line 16). Finally the created WAPD containing all control dependencies is returned (line 19).

```
1.  Algorithm createControlDependency
2.  Input:  wapd : A WAPD
            v : the start node of a WebParseTree
3.  Output: wapd : A WAPD with Control Dependencies
4.  Begin
5.    v.isVisited() := true;
6.   // First time creating wapd
7.   if (wapd == null) {
8.      wapd := createWAPD ();
9.      wapd.setEnteringNode (v);
10.  }
11.
12.  //Recursively create Control Dependencies
13.  foreach( w ∈ v.getAllChilds() ) {
14.   If (w.isNotVisited()){
15.     wapd.buildControlDependency (v,w);
16.     wapd := createControlDependency(wapd,w);
17.  }
18. }
19.   return wapd;
```

Figure 11. Create Control Dependencies on WAPD from WebParseTree.

The resulting WAPD is further processed and enhanced by data and event dependencies. This will be explained in the next sections.

4.2.2 Create event dependencies

An event dependency can be created by linking a dependency on an event source to an event sink. An event source fires an event according to its event handlers. On the other hand, an event sink is the target point called by an event source.

Node (20) of $WebParseTree$ in Figure 8 is an event source with a HTML input submission type $<input\ type = "submit"\ value = "login"\ name = "fsubmit">$. If the event $Onclick$ is trigged on this event source, the event sink on node 5 is executed. This behavior is modelled in the WAPD shown in Figure 12 (a).

The event dependency modeled by means of an object diagram is illustrated in Figure 12 (b). An event source E_H on node (20) is associated with event sink E_C on node (5) via D_E. Every event dependency has to have a *Label* with a trigger event

Journal of Theoretical and Applied Information Technology
20th February 2016. Vol.84. No.2

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

associated with it. As a result, the D_E dependency is labelled by an event *onClick* ($L_{onClick}$).

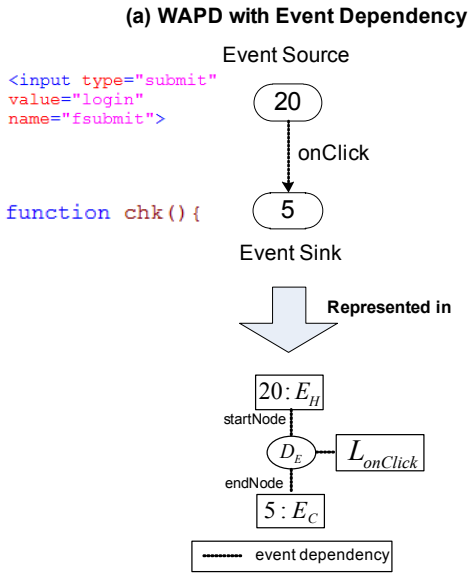

Figure 12: Example of building Event Dependency with Label

Figure 13 shows the algorithm to create event dependencies. It takes a WAPD with control dependencies as input (the result of the previous step). At first, every node in wapd is scanned and looked for an event source (line 5). If an event source is found (line 6), the *EventDependency* is added to the wapd by mapping the source node as event source (line 8) and the destination node as event sink (line 9). An event handler is assigned to a Label (line 10) associated to an event dependency (line 11). Finally, an event dependency is created and added to the wapd (line 12). The algorithm returns a wapd with control- and event dependencies.

```
1   Algorithm createEventDependency
2   Input:  wapd : A WAPD with Control Dependencies
3   Output: wapd : A WAPD with Control & Event
                   Dependencies
4   begin
5    foreach ( node ∈ wapd ) {
6      if (node.containsEventSource())
7        EventDependency evtDep := createEventDep();
8        Node evtSrc:= node.getEventSource(wppd);
9        Node evtSink:= node.getEventSink(wppd);
10       Label lb := evtSrc.getEventTrigger();
11       evtDep.addLabel(lb);
12       wapd.addDependency(evtDep , evtSrc, evtSink);
13     }
14   }
15  return wapd;
16  end;
```

Figure 13. Create Event Dependencies on WAPD.

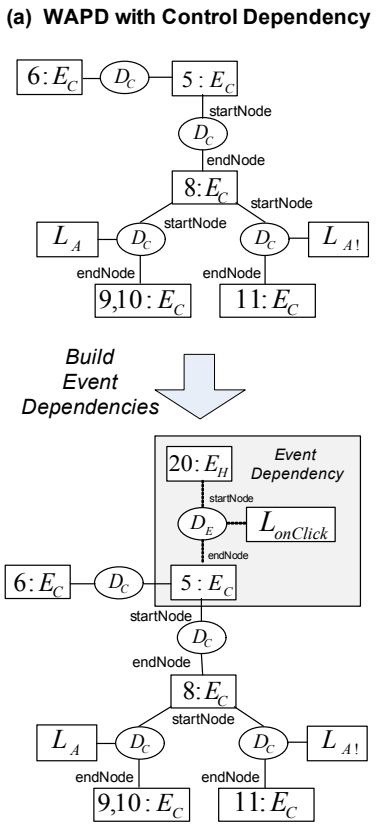

Figure 14. Create Event Dependencies by using an algorithm.

The example of creating event dependencies is shown in Figure 14. The WAPD model (see Figure 9 and Figure 10) containing control dependencies from node (5) to (6) and (8) is used as input for the *createEventDepedency* algorithm. As node 20 is an event source and node 5 is an event sink, an event dependency (D_E) from node 20 ($20: E_H$) to node 5 ($5: E_C$) is added to the wapd associated with an *onClick* label (i.e., $L_{onClick}$). The algorithm returns the enhanced wapd now containing control- and event dependencies as shown in Figure 14(b).

4.2.3 Create data dependencies

Data dependencies are used to express how data flows inside a program [24]. A data dependency as introduced in [25] is created by associating a start node defining a variable (*def*) and end node defining its usage (*use*).

An example of creating a data dependency on the WebParseTree in Figure 8 is shown in Figure 15 (a). A def is identified in node (18) specifying an HTML input form ($< input\ type = "text"\ name = "user" >$). The respective use is

Journal of Theoretical and Applied Information Technology
20th February 2016. Vol.84. No.2

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

identified in node 6 as a user variable to store the entered value. The corresponding object diagram containing the data dependency is shown in Figure 15 (b). There is no *Label* associated with D_D as data can be used directly from def to use without any guard condition.

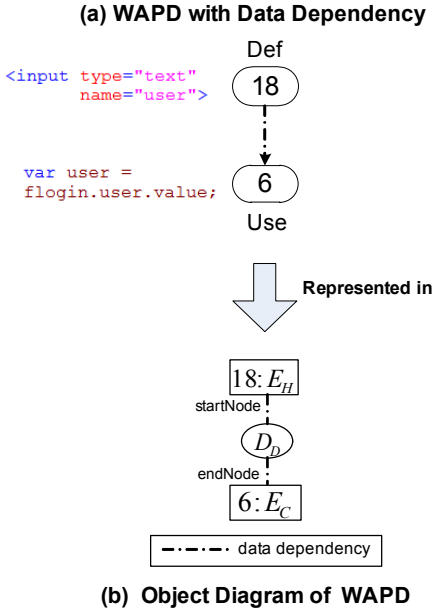

Figure 15: Example Of Building Data Dependency.

Our algorithm to create data dependencies is presented in Figure 16. It takes a WADP with control- and event dependencies from the prior step as input. The algorithm scans every node of the wapd (line 5) and searches for variable definitions, called *def node* (line 6). This might be variable initializations in a server/client side script or a HTML form. If a *def* node is found, the algorithm scans for every *use* node (line 9) and assign each *use* node as a destination node. Normally, a *use* node may have more than one associated def node. Each data dependency is added to the wapd (line 11).

To illustrate the creation of data dependencies our example WAPD model in Figure 14(b) is taken as input. A variable (i.e., HTML input object) is defined in node (18), and it is referred by a client-side script in node (6). Hence, a data dependency (D_D) is created which links from node 18 (18: E_H) to node 6 (6: E_C) as shown in Figure 17.

```
1   Algorithm createDataDependency
2   Input:  wapd : A WAPD with Control & Event
3   Output: wapd : A complete WAPD with Control,
    Event & Data Dependency
4   begin
5     foreach (node ∈ wapd ) {
6       if (node.containsDefVariable()){
7          Node  defNode= node.getDefVariable(wapd);
8          Node [] useNode= node.getUseVariable(wapd);
9          foreach(node ∈ useNode){
10          DataDependency dataDep:= createDataDep();
11          wapd.addDependency(dataDep , def, node);
12         }
13      }
14    }
15  return wapd;
16  end;
```

Figure 16: Create Data Dependencies on WAPD.

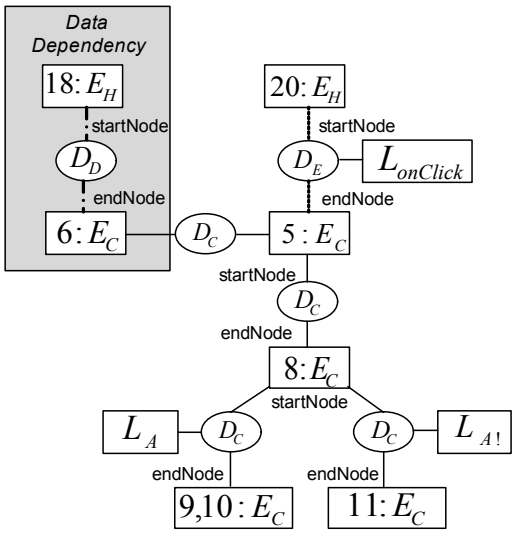

Figure 17. An Example Of Creating Data Dependencies Using A Createdatadependency Algorithm.

4.2.4 Putting the steps together

In the last sections we have step by step presented how to transform the source code of a web application to our WAPD model. This models abstracts from the implementation details and stores all information regarding the control flow, the data flow and the events by means of dedicated dependency types. Figure 18 and Figure 20 depicts the resulting WAPDs of the pages Login.html and LoginAction.php as UML object diagrams. It can be seen that data and event dependencies can link respective nodes across webpages. For example, the data from node (18: E_H) and node (19: E_H) of page *Login.html* flows to node (2: E_S) and node (3: E_S) of page *LoginAction.php* respectively.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

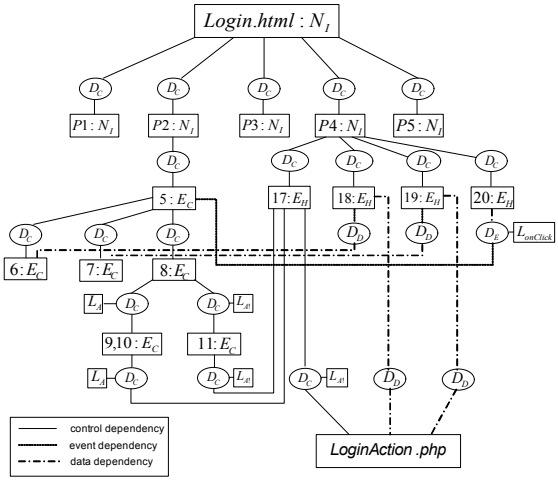

Figure 18: WAPD of Login.Html Represented By An Object Diagram.

5. COMPARISON WITH EXISTING APPROACHES

A lot of criteria have to be concerned when we model web applications. Unfortunately, there are no standard that we can use as referential criteria. Base on literatures that we have reviewed, and to the best of our knowledge, these following criteria should be considered when modeling web applications:

- Structural Analysis: this is a basic requirement on modeling web applications. There are two levels on structural analysis of web applications: (1) page level analysis and (2) code level analysis. Page level analysis focuses only on link relationship between webpages while code level analysis analyzes source codes. An ideal web application model definitely consists of both page and code level analysis.
- *Page Level Analysis*: HTML tags that can product a request to another webpage such as hyperlink (< a >) and submission form (< Form >) are analyzed. These relationships are used to model links between webpages.
- *Code Level Analysis*: this analyses source codes which contains three programming parts, i.e., HTML, *Server-Side Scripts* and *Client-Side Scripts*. A completed model must store all of these three programming information as proposed in [11] including our proposed WAPD model. However, some web application models focus on analyzing only HTML and server-side scripts. These models are proposed in [6], [7], [10]–[15]. Likewise, some models focus on analyzing HTML and client-side scripts proposed in [8], [9].
- *Automatic Approach*: models can be built automatically by providing methodologies such as models proposed in [2], [7]–[9], [17]. However, some models are built manually which require well-educated people to create models. In spite of using manual approach, an automatic approach is more practical when modeling web applications.
- *Extended to generate test cases*: a model can be extended to produce test cases, and can be used in other purposes such as code transformation.

Functional / Model	Structural Analysis	Page Level Analysis	Server-Side Scripts	Client-side Scripts	Automation Approach	Generate Test Case
WAPD	◆	◆	◆	◆	◆	◆
Ricca [2]	◆	◆	◆	◇	◇	◆
Reza [3]	◆	◆	◇	◇	◇	◆
Rafique [4]	◆	◆	◇	◇	◇	◆
Machra [5]	◆	◆	◇	◇	◇	◆
Youxin [6]	◆	◆	◆	◇	◇	◆
Wassermann [7]	◆	◆	◆	◇	◆	◆
Artzi [8]	◆	◆	◇	◆	◆	◆
Mesbah [9]	◆	◆	◇	◆	◆	◆
Dia [10]	◆	◆	◆	◇	◇	◆
Ricca [11]	◆	◆	◆	◆	◇	◆
GU [12]	◆	◆	◆	◇	◇	◇
Tung [13]	◆	◆	◆	◇	◇	◆
Sabharwal [14]	◆	◆	◆	◇	◇	◆
Bansal [15]	◆	◆	◆	◇	◇	◆
Achkar [16]	◆	◆	◇	◇	◇	◆
Garcia [17]	◆	◆	◇	◇	◆	◆

◆= *Supported* ◇= *Not Supported*

Figure 19. A Comparison of web applications modeling.

Figure 19 lists the web application models proposed by authors mentioned in section 2. The Figure compares models with the criteria described above. To the best of the author's knowledge, this WAPD model supports all the criteria that is necessary for modeling web applications.

Journal of Theoretical and Applied Information Technology
20th February 2016. Vol.84. No.2

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

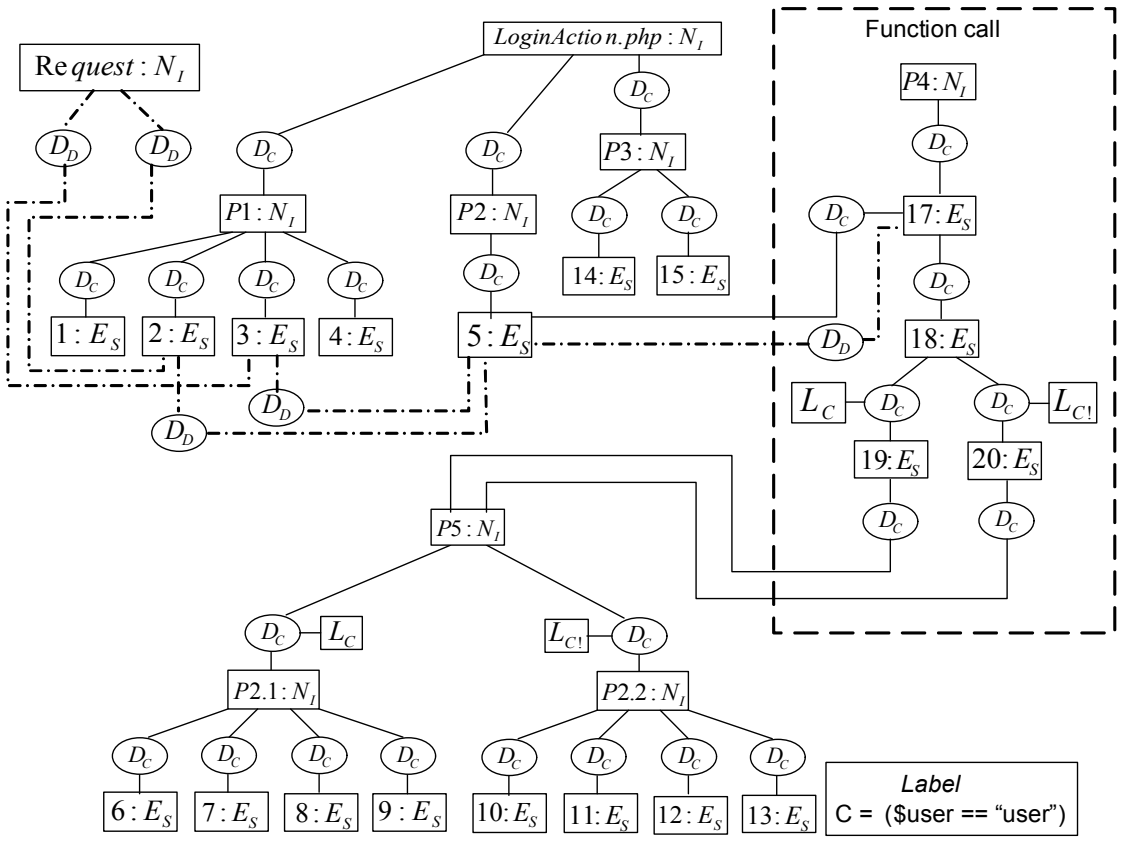

Figure 20: WAPD of LoginAction.php represented by an object diagram.

6. CONCLUSION AND FUTURE WORK

We have introduced an approach to automatically transform the source code to WAPD model. A WebParseTree is transformed from source codes by WA parser. Dependencies and expressions on WebParseTree are analyzed by WA generator, and transformed to WAPD. The WAPD stores structure and behaviors of web applications. This generic abstraction model can be extended to be used for many purposes such as (1) code generation for generating source code into a certain language. This code generation concept will analyze information from the model and generate new source code which is known as code transformation and (2) test generation for generating test cases analyzed from the model. For our future work, we are going to analyze the model in order to generate test cases in terms of white-box testing.

REFERENCES

[1] P. Jorgensen, *Software testing: a craftsman's approach*. CRC Pr I Llc, 2002.

[2] F. Ricca and P. Tonella, "Analysis and testing of Web applications," pp. 25–34, Jul. 2001.

[3] H. Reza, K. Ogaard, and A. Malge, "A Model Based Testing Technique to Test Web Applications Using Statecharts," *Fifth Int. Conf. Inf. Technol. New Gener. (ITNG 2008)*, pp. 183–188, Apr. 2008.

[4] N. Rafique, N. Rashid, S. Awan, and Z. Nayyar, "Model Based Testing in Web Applications," *Int. J. Sci. Eng. Res.*, vol. 2, no. 1, pp. 56–60, 2014.

[5] S. Machra and N. Khatri, "Model Based Testing of Website," *Int. J. Comput. Sci. Appl.*, vol. 4, no. 1, pp. 143–152, 2014.

[6] M. Youxin, W. Dafa, and D. Junwei, "Research on Framework of Test Case Generation of Web Applications Based on Z Specification," *2009 Int. Forum Inf. Technol. Appl.*, pp. 555–558, May 2009.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

[7] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su, "Dynamic test input generation for web applications," in *Proceedings of the 2008 international symposium on Software testing and analysis - ISSTA '08*, 2008, p. 249.

[8] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip, "A framework for automated testing of javascript web applications," in *Proceeding of the 33rd international conference on Software engineering - ICSE '11*, 2011, p. 571.

[9] A. Mesbah and I. C. Society, "Invariant-Based Automatic Testing of Modern Web Applications," *Computer (Long. Beach. Calif).*, vol. 38, no. 1, pp. 35–53, 2012.

[10] Z. Dai and M.-H. Chen, "Automatic Test Case Generation for Multi-tier Web Applications," *2007 9th IEEE Int. Work. Web Site Evol.*, pp. 39–43, Oct. 2007.

[11] P. Tonella, F. Ricca, and P. Trento, "A 2-layer model for the white-box testing of Web applications," in *Web Site Evolution, Sixth IEEE International Workshop on (WSE'04)*, pp. 11–19.

[12] J. Gu, L. Xu, B. Xu, and H. Yang, "An Extended MM-Path Approach to Component-Based Web Application Testing," in *2008 12th IEEE International Workshop on Future Trends of Distributed Computing Systems*, 2008, vol. 2, pp. 144–150.

[13] Y.-H. Tung, S.-S. Tseng, T.-J. Lee, and J.-F. Weng, "A Novel Approach to Automatic Test Case Generation for Web Applications," *2010 10th Int. Conf. Qual. Softw.*, pp. 399–404, Jul. 2010.

[14] S. Sabharwal, "Modeling the Navigation Behavior of Dynamic Web Applications," *Int. J. Comput. Appl.*, vol. 65, no. 13, pp. 20–27, 2013.

[15] P. Bansal and S. Sabharwal, "A model based approach to test case generation for testing the navigation behavior of dynamic web applications," in *2013 Sixth International Conference on Contemporary Computing (IC3)*, 2013, pp. 213–218.

[16] H. Achkar, "Model Based Testing of Web Applications," *Proc. 9th Annu. STANZ, Aust.*, pp. 1–28, 2010.

[17] B. García and J. C. Dueñas, "Automated Functional Testing based on the Navigation of Web Applications," *Electron. Proc. Theor. Comput. Sci.*, vol. 61, pp. 49–65, 2011.

[18] M. Weiser, "Program slicing," pp. 439–449, Mar. 1981.

[19] B. Korel and C. Science, "The program dependence graph in static program testing," *Inf. Process. Lett.*, vol. 24, no. 2, pp. 103–108, Jan. 1987.

[20] L. D. Fosdick and L. J. Osterweil, "Data Flow Analysis in Software Reliability," *ACM Comput. Surv.*, vol. 8, no. 3, pp. 305–330, Sep. 1976.

[21] J. Warmer and A. Kleppe, *The Object Constraint Language: Precise Modeling with UML*, 2nd ed. Addison-Wesley, 2003.

[22] A. V. Aho, "Algorithms for finding patterns in strings," *Handbook of Theoretical Computer Science, volume A: Algorithms and Complexity*. The MIT Press, pp. 255–300, 1990.

[23] F. E. Allen, "Control Flow Analysis," *Proc. ACM Symp. Compil. Optim.*, pp. 1–19, 1970.

[24] M. S. Hecht, *Flow Analysis of Computer Programs*. North-Holland, 1977.

[25] S. Rapps and E. J. Weyuker, "Selecting Software Test Data Using Data Flow Information," *IEEE Trans. Softw. Eng.*, vol. SE-11, no. 4, pp. 367–375, 1985.