DHTML Accessibility Checking Based on Static JavaScript Analysis

Takaaki Tateishi, Hisashi Miyashita, Tabuchi Naoshi, Shin Saito, and Kouichi Ono

Tokyo Research Laboratory, IBM Research
1623-14 Shimotsuruma, Yamato-shi, Kanagawa-ken, 242-8502 Japan
{tate,himi,tabee,shinsa,onono}@jp.ibm.com

Abstract. DHTML accessibility is being standardized by W3C, which provides metadata for UI widgets commonly implemented by HMTL and JavaScript. However it is difficult to check that webpages always have correct metadata according to the standards of DHTML accessibility since UI widgets can be updated by JavaScript programs. Thus we propose a technique for checking accessibility of UI widgets. In this check, we use static program analysis techniques so that we can check accessibility without executing a program. In addition, we developed a prototype system based on the proposed technique and applied it to a simple DHTML application.

Keywords: DHTML accessibility, JavaScript, Static program analysis.

1 Introduction

Rich internet applications based on DHTML are spreading globally. One key component is a client-side scripting language, JavaScript in the most common case. Such a JavaScript program can operate directly on the internal tree representation of a webpage called a DOM (Document Object Model) instance, and contributes to enhance both the functionality and usability of dynamic Web applications. One of the popular methods to enhance the functionality and usability is to create UI widgets such as checkboxes and tab panels. Such UI widgets are represented by a DOM instance and JavaScript programs called event handlers. However a lot of UI widgets are inaccessible since they don't provide information about what is represented by DOM instances, and thus a screen reader cannot identify them as UI widgets. Consequently people with visual disabilities cannot understand what UI widgets are on a webpage.

DHTML accessibility [13, 14, 15] is being standardized by W3C to address such situations, and some of these features are supported by Firefox [5]. The key concept of the standards is a set of accessibility metadata, called accessibility roles and states, for identifying classes of UI widgets such as Checkbox and those valid accessibility states such as "checked" or "unchecked". It also supplies rules for state changes to specify how UI widgets should behave. Assistive technologies such as screen readers can provide useful information for people with visual disabilities by interpreting the accessibility metadata. Thus proper accessibility metadata should be a part of HTML

C. Stephanidis (Ed.): Universal Access in HCI, Part III, HCII 2007, LNCS 4556, pp. 167–176, 2007.

documents so that people with visual disabilities can understand the structures of the UI widgets and their states.

For the development of DHTML applications, developers should write event handlers for UI widgets that operate on DOM instances according to the DHTML accessibility standards. One of the methods to check accessibility metadata is validation of an HTML document against the standards, but this is not adequate, since we actually need to check the DOM instance each time it is updated by the program, and also check the behavior of the UI widgets. In addition, testing is a popular technique to verify the behavior of software applications as well as dynamic Web applications, but it is hard to check all of the possible DOM instances updated by the program.

In this paper, we propose a novel technique to check that an UI widget behaves according to the DHTML accessibility standards by using a static program analysis technique called DOM analysis [7] to assist developers in writing accessible DHTML applications. Static program analysis involves techniques for predicting values or approximations to the set of possible results that will arise at run-time without running a program. A number of approaches to static program analyses are being studied for different purposes. We also developed a DOM analysis for the purpose of predicting the set of DOM instances updated by a JavaScript program.

With this technique, we formally define a constraint on the accessibility metadata and state changes, called an accessibility rule, for each UI widget based on the DHTML accessibility standards. Such an accessibility rule is described with a state transition diagram and schemata. A state transition diagram consists of states and state transitions, and specifies how the accessibility states are changed. Each schema is associated with a state of the state transition diagram and defines the validity of the roles and states such as the valid states for the roles and the parent-child relationships between roles.

This paper is organized as follows: Section 2 describes the accessibility roles and states with a simple example, and then formalizes accessibility of UI widgets for DHTML applications. In Section 3 we explain how to check the accessibility of UI widgets against the accessibility rules. In Section 4 we comment on our experiments with a prototype system. In Section 5 we briefly describe some related techniques and tools for checking DHTML accessibility. In Section 6 we conclude this paper.

2 DHTML Accessibility and Its Formalization

In this section, we first explain the accessibility roles and states and then formally define the accessibility of an UI widget using a state transition diagram and schemata.

2.1 UI Widgets and Accessibility Metadata

Accessibility roles and states are added to an HTML document as XML attributes for identifying classes of UI widgets and their valid accessibility states. By interpreting the accessibility roles and state, assistive tools such as a screen reader can recognize the HTML element as an UI widget having a valid accessibility state.

Figure 1 is a HTML fragment representing a checkbox. The appearance of the checkbox is defined by the class attribute specifying a style sheet. The x2 and state are namespaces, defined in the drafts of W3C standards [14, 15], for accessibility roles and states respectively.

```
<span class="checkbox" id="chbox1" x2:role="role:checkbox"
      state:checked="true" tabindex="0"
      onkeydown="return checkBoxEvent(event);"
      onclick="return checkBoxEvent(event);" >
a checkbox sample</span>
```

Fig. 1. A HTML fragment representing a checkbox widget

In addition, behavior of the checkbox is implemented by a JavaScript program specified by the onkeydown attribute and the onclick attribute as shown in Figure 2. The program reacts to user events of mouse click and key down, and changes the accessibility state of the checkbox with the setAttributeNS function.

```
function checkBoxEvent(event){
  if ((event.type == "click" && event.button == 0) ||
       (event.type == "keydown" && event.keyCode == 32)) {
    var checkbox = event.target;
    if (checkbox.getAttributeNS("http://www.w3.org/2005/07/aaa", "checked")
        == "true") {
      checkbox.setAttributeNS("http://www.w3.org/2005/07/aaa" "checked",
                              ”checked”,”false”);
    } else {
      checkbox.setAttributeNS("http://www.w3.org/2005/07/aaa",
                              "checked",”true”);
    }
    return false;
  }
  return true;
};
```

Fig. 2. An event handler for the checkbox widget

Such a UI widget has a set of constraints on its accessibility states and state changes called an accessibility rule. A constraint on the accessibility states determines how to specify the accessibility state of a UI widget. For example, a checkbox widget must have a checked attribute and its value must be true or false. In addition, from those valid states we can define a constraint on state changes. For example, if there is a click event and the current value of a checked attribute is true, it should be changed to false. Otherwise it should be changed to true. We call such a set of constraints an accessibility rule. In the next section, we formalize the accessibility rule.

2.2 Formal Description for Accessible UI Widgets

We define an accessibility rule for each UI widget using a state transition diagram and a schema language playing a similar role with DTD based on regular expression type [3, 4] to describe the behavior of the UI widget and constraints on accessibility state respectively. In addition, the schema language is equivalent to the regular tree grammar [2].

For example, the following schema represents a checked checkbox where each attribute is considered as a label with prefixed @ and String represents an arbitrary string.

```
Checkbox → span[Id, Class, Checked]

Id       → @id[String]

Class    → @class[String]

Checked  → @checked[true]
```

This schema is corresponding to the following DTD definition.

```
<!ELEMENT  span  EMPTY!>
<!ATTLIST  span  id        #PCDATA!>
<!ATTLIST  span  class    #PCDATA!>
<!ATTLIST  span  checked  true!>
```

The syntax of the schema language is defined as follows.

```
Schema  := Rule, Rule, …
Rule     := Var → Tree
Tree     := Label [ Tree ]
          | Tree,Tree
          | Tree|Tree
          | Var
          | ()
          | {}
```

A schema consists of a set of rules. Each rule constrains the structure of a tree with a pattern Tree and is referred to by Var so that the pattern can be used in other rules. Such a pattern can represent a tree labeled by Label, a sequence of trees separated by "," or a choice of trees by "|", where the label represents HTML elements or attributes. In addition, "()" represents an empty sequence and "{}" means that there is no choice. An arbitrary number of trees is represented by "*", and it is defined as follows.

```
Var → Tree*  =  Var → Tree Var | ()
```

In our accessibility rules, the schema of the checked checkbox is generally defined by the following schema where Any represents an arbitrary element and A represents all the attributes except checked.

```
Checkbox → Any[A*, Checked, A*]

Checked  → @checked[true]

A        → (@id[String]|@class[String]|…)
```

If we define a schema for an unchecked checkbox, we use false instead of true in the rule for Checked.

The behavior of a UI widget is defined by a state transition diagram that consists of states and transitions. A transition has an event and a guard condition to determine when an accessibility state is changed. With a state transition diagram and schemata, we define the accessibility rules of UI widgets in which each state of a state transition diagram is associated with a schema representing an accessibility state.

Figure 3 shows an example of an accessibility rule for the checkbox. In the accessibility rule, CheckedSchema is a schema of the checked checkbox and it is associated with the checked state. UncheckedSchema is a schema of the unchecked checkbox and it is associated with the unchecked state. Each transition says that a checkbox changes its state when a user clicks the widget using a left button which is represented by 0 in JavaScript.

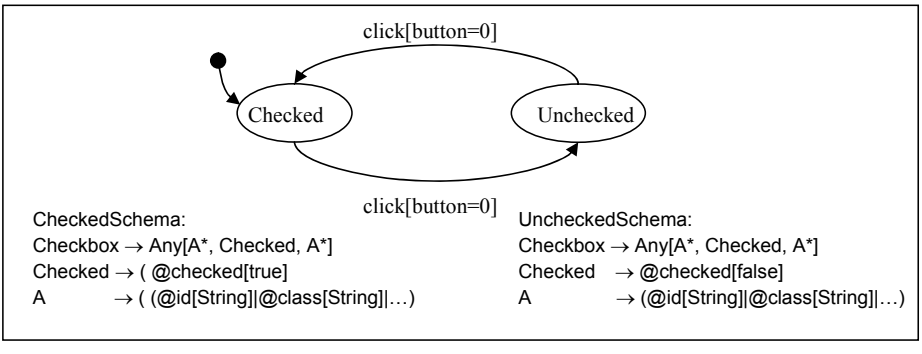

Fig. 3. State transition diagram representing behavior of the checkbox

3 Checking DHTML Accessibility

In this section, we describe how to check the accessibility of a UI widget against a corresponding accessibility rule. The main idea for this check is to verify that the UI widget is always accessible. More properly, it satisfies the corresponding schema as long as it behaves according to the corresponding state transition diagram. However the DOM instance representing a UI widget can be updated without limit and we cannot check all of the DOM instances. Therefore we focus on repeated updates and omit the checks against the repeated updates. Consider the example of the checkbox. The state of the checkbox is changed to checked and unchecked alternately as shown in Figure 4. Thus if we verify that (1) any DOM instance representing a checked checkbox is transformed to one representing a unchecked checkbox and (2) any DOM instance representing a unchecked checkbox is transformed to one representing a checked checkbox, we can omit the verifications of the other same transitions. This leads to the conclusion that we can divide the verification against the state transition diagram into one for each state transition, and check that each DOM instance satisfying a schema associated with a pre-state is transformed to one satisfying a

schema associated with a post-state. If all of the verifications of state transitions are successful, then the target UI widget satisfies the accessibility rule.

In order to describe the verification more precisely, we first briefly explain the static program analysis techniques used by the verification, and then describe how to use those techniques for the verification.

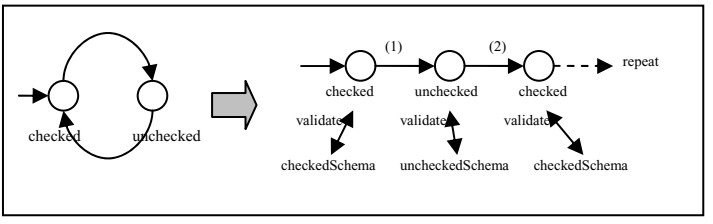

Fig. 4. Verifying an UI widget against an accessibility rule

3.1 Static Program Analyses

In the verification, we use the two complementary static program analysis techniques called dynamic slicing [8] and DOM analysis [7].

Dynamic Slicing

The dynamic slicing is a technique for extracting statements affecting the values of variables for particular input data. A collection of extracted statements is called a slice. As a simple example, consider a program shown in Figure 5 with input data event.type, event.button and event.target such that event.type is “click”, event.button is 0 and event.target is a DOM element that has a checked attribute with the value “true”. The deleted (struck-out) statements do not affect the DOM elements “checkbox” and we can extract the other statements as a slice.

```
if (event.type == "click" && event.button == 0) {
  var checkbox = event.target;
  if (checkbox.getAttribute("checked") == "true") {
     checkbox.setAttribute("checked", "false");
  } else {
     checkbox.setAttribute("checked","true");
  }
 return false;
}
return true;
```

Fig. 5. A slice of a JavaScript program

DOM Analysis

DOM analysis is a technique for inferring DOM instances as updated by a program. Such inferred DOM instances can be represented by a schema. Thus we can check that the inferred DOM instances are accessible by checking that they are included in the schema of an accessibility rule. For example, if we have a set of DOM instances

represented by the CheckedSchema schema of Figure 3 and infer the DOM instances as updated by the program of Figure 5, we can obtain a set of DOM instances represented by the following schema that represents a checked checkbox and an unchecked checkbox.

```
Checkbox → Any[A*, Checked, A*]

Checked  → @checked[true|false]

A        → (@id[String]|@class[String]|…)
```

In addition, if we use the sliced program shown in Figure 5, we can obtain a set of DOM instances represented by the following schema that represents only an unchecked checkbox.

```
Checkbox → Any[A*, Checked, A*]

Checked  → @checked[false]

A        → (@id[String]|@class[String]|…)
```

Those schemata can be compared with other schema such as CheckedSchema and UncheckedSchema of Figure 5 using the same algorithm as for regular expression types.

3.2 Verifying the Accessibility of UI Widgets

In order to verify the accessibility of a UI widget on a dynamic webpage against a corresponding accessibility rule, we first extract a DOM element providing a view of each UI widget and all event handlers that update the DOM instance (as described in Step 1), and then we check that the UI widget always satisfies a corresponding schema (in Step 2 and 3).

Step 1: Extract an UI widget and its event handlers
Accessibility roles are the key metadata for extracting UI widgets. We consider a DOM element having an accessibility role as a UI widget since assistive tools recognize classes of UI widgets based on the accessibility roles. In addition, we identify event handlers based on attributes such as onclick and onkeydown. For example, Figure 1 and Figure 2 are an extracted UI widget and the event handlers respectively, since the span element has the checkbox role and the onclick and onkeydown attributes are specified in the JavaScript program of Figure 2.

Step 2: Check that the initial DOM instance satisfies a schema
The main idea for checking accessibility of a DOM instance that could be updated without limit is to divide the verification against its corresponding accessibility rule into verifications against each state transition. This inductive idea is valid if the initial DOM instance is accessible. Thus we have to verify that the initial DOM instance extracted in Step 1 is accessible by comparing the DOM instance with the schema of the initial state. For the checkbox widget, we check if the DOM instance described in Figure 1 satisfies the CheckedSchema schema described in Figure 3.

Step 3: Check that every update is legal according to a corresponding state transition
We divide verification against an accessibility rule into verification against each state transition as described above. In the verification against a state transition, we verify

that an accessible DOM instance is transformed into another accessible one with a corresponding event handler.

For example, when we verify that the checkbox described in Figures 1 and 2 is updated according to the state transition from the checked state to the unchecked state shown in Figure 3, we can extract the same slice as in Figure 5. This is because the state transition has a click event and the guard condition button=0 and thus we can use "click" and 0 for event.target and event.button respectively. Next we can infer a set of DOM instances from the CheckedSchema schema. The set of the inferred DOM instances are represented by the following schema.

```
Checkbox → Any[A*, Checked, A*]

Checked  → @checked[false]

A        → (@id[String]|@class[String]|…)
```

Finally we check that the set of the inferred DOM instances is included in the UncheckedSchema schema.

For this verification, we perform the following three steps:

1. Extract a slice corresponding to the target state transition from the event handler. In order to extract a slice corresponding to a target state transition, we use an event, a guard condition and a schema associated with a pre-state to identify the input data.
2. Infer a set of updated DOM instances from a set of DOM instances that satisfies the schema associated with the pre-state of the state transition by applying DOM analysis to the extracted slice. Note that the set of the inferred DOM instances contains all of DOM instances arising at run-time and it is represented in the schema language.
3. Check that the set of the updated DOM instances is included in the corresponding schema associated with the post-state of the state transition.

4 Experiments

We developed a prototype of a DHTML accessibility checker based on the proposed technique. The prototype has prior knowledge of the accessibility rules corresponding to the UI widgets. Thus developers need not prepare any accessibility rules by themselves.

We used the prototype for checking the accessibility of a checkbox in accessible DHTML samples provided by the Mozilla.org project against three types of the accessibility rules shown in Figures 3, 6 and 7, testing on a laptop PC with a Pentium-M 2.13GHz CPU and 1.5GB of memory. The time for the slicing steps (Step 3-1), the inference steps (Step 3-2), the comparison steps (Step 3-3) and the total time are shown in Table 1.

As shown in the results, most of time was taken by the comparison phases, since we employed only the basic algorithm for comparing regular expression types though there are some high-level techniques to improve performance. In addition, theoretical total time for inference phases and comparison phases proportionally increases according to the number of state transitions. This prospect is shown in the results of Figure 3 and 6. However, the inference time and the comparison time for Figure 7 is worse than Figure 3 since an extracted slice and a set of inferred DOM instances for Figure 7 become larger or more complex than Figure 3.

Table 1. Execution time of the verification

accessibility rule	slicing time (ms)	inference time (ms)	comparison time (ms)	total time (ms)
Figure 3	234	516	3860	7391
Figure 6	251	953	6186	10140
Figure 7	203	750	24625	28266

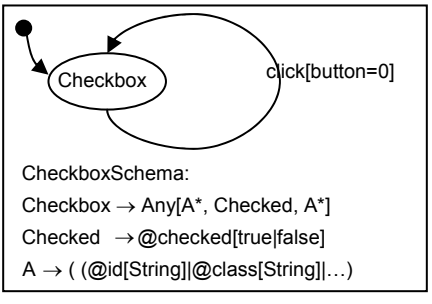

Fig. 6. An accessibility rule with one state

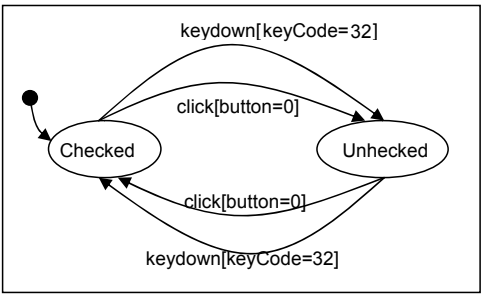

Fig. 7. An accessibility rule with the keydown event

5 Related Work

There are a lot of automated accessibility check tools [16] such as WebXACT [17] or WebKing [9] and related research activity. As for WebKing, it provides functionality for checking dynamically constructed webpages. For this check, it requires scripts in which the tester describe how a user reacts to the webpage, such as by filling out an input form with text, and then it checks the pre-defined rules, including the accessibility by executing the applications. On the other hand, our proposed technique does not need to execute so it is applicable to accessibility checking during the development phase.

From the viewpoint of the flexibility of tools, a concept of separation between checking rules (i.e.: standards and guidelines) and rule engines (i.e.: system for evaluating the checking rules) is important. Some tools such as Lift [12] allow us to customize a guideline according to specific requests such as a corporate guideline by selecting pre-defined rules. In order to make a customization more flexible, a language for defining guidelines and a framework for evaluating custom guidelines was proposed [1, 6]. We can describe static features of guidelines using the guideline definition language, but we can not describe dynamic features such as the behavior of the UI widget.

In addition, Sun et al. [11] and Pontelli et al. [10] focus on semantic features of webpages for improving navigability. Sun uses automata for modeling online web transactions as we use state transition diagrams for modeling the behavior of the UI

widgets. Pontelli provides a language for providing semantic description of tables found in web pages and for navigation strategies.

6 Conclusion

In this paper, we proposed a fundamental technique for verifying the accessibility of UI widgets in a DHTML application during the development phase. In this verification, we check that an UI widget always has proper accessibility state according to a corresponding accessibility rule. In addition, we use power of static program analysis techniques so that we can perform the accessibility checking without execution. In the future, we will improve our prototype so as to apply it to a set of UI widgets implemented as a part of JavaScript libraries such as DOJO.

References

1. Beirekdar, A., Keita, M., Noirhomme, M., Randolet, F., Vanderdonckt, J., Mariage, C.: Flexible Reporting for Automated Usability and Accessibility Evaluation of Web Sites. In: Proc. of Human-Computer Interaction (2005)
2. Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree Automata Techniques and Applications http://www.grappa.univ-lille3.fr/tata
3. Hosoya, H., Vouillon, J., Pierce, B.: Regular expression types for XML. In: Proc. of the International Conference on Functional Programming, pp. 11–22 (2000)
4. Hosoya, H., Pierce, B.: Regular expression pattern matching for XML. In: Proc. of Principles of Programming Languages, pp. 67–80 (2001)
5. Mozilla: Accessible Rich Internet Application http://developer.mozilla.org/en/docs/Accessible_DHTML
6. Vanderdonckt, J., Beirekdar, A.: Automated Web Evaluation by Guideline Review. Journal of Web. Engineering 4(2), 102–117 (2005)
7. Tateishi, T., Miyashita, H., Saito, S., Ono K.: Automated Verification Tool for DHTML. In: Proc. of Automated Software Engineering (2006)
8. Tip, F.: A survey of program slicing techniques, Journal of Programming Languages, pp. 121–181 (1995)
9. Parasoft: WebKing, http://www.parasoft.com/jsp/products/home.jsp?product=WebKing
10. Pontelli, E., Xiong, E., Gupta, G., Karshmer, A.I.: A Domain Specific Language Framework for Non-Visual Browsing of Complex HTML Structures. In: Proc. of the International Conference on Assistive technologies (2000)
11. Sun, Z., Mahmud, J., Mukherjee, S., Ramakrishnan, I.V.: Model-directed web transactions under constrained modalities. In: Proc. of the International Conference on World Wide Web, pp. 447–456 (2006)
12. UsableNet Inc.: Lift for Dreamweaver http://www.usablenet.com/products_services/lift_dw/lift_dw.html
13. W3C: Dynamic Accessible Web Contents Roadmap http://www.w3.org/WAI/PF/roadmap/
14. W3C: Roles for Accessible Rich Internet Applications http://www.w3.org/TR/aria-role/
15. W3C: States and Properties Module for Accessible Rich Internet Applications http://www.w3.org/TR/aria-state/
16. W3C: Web Accessibility Evaluation Tools http://www.w3.org/WAI/ER/tools/
17. Watchfire Corporation: WebXACT http://webxact.watchfire.com/