Chapter 6

Java Server Pages Reverse Engineering

6.1 JSPick - A Java Server Pages Design Recovery Tool

JSPick [127][66] is a reverse engineering tool for Java Server Pages based presentation layers. JSPick allows for automatic generation of a documentation of the whole system interface in an easy to read specification language.

Consider javadoc, the standard Java documentation tool. It is not suitable for documenting Java Servlets[1], because applying javadoc to the customized server script classes leads only to the documentation of the technical parameters HTTPRequest and HTTPResponse. The documented technical signatures are not amenable for a reasonable application of proposed specification techniques like e.g. design by contract [123][124][182]. The interesting parameters, which are significant for the functional requirements and the business logic, namely the HTML form parameters provided by the forms or links calling the page, cannot be documented automatically. All this amounts to say that javadoc is a pure redocumentation tool in the sense of [29]. The generated code documentation adds no value to the semantics that can be read directly from the code, only a visualization is created.

In contrast, JSPick is a design recovery tool in the sense of the reverse engineering seminal paper [29]. JSPick exploits the Next Server Pages concepts in order to infer from the code a meaningful documentation at a higher abstraction level. JSPick extracts from a system interface all pages with their signatures together with the contained links and forms. JSPick generates a GUI browser for a given system. With that browser the developer can examine source code,

[1]Anyway javadoc would not be applied to the Java Servlets code that is generated by a JSP container. However the considerations concerning javadoc foster the understanding of the JSPick concepts.

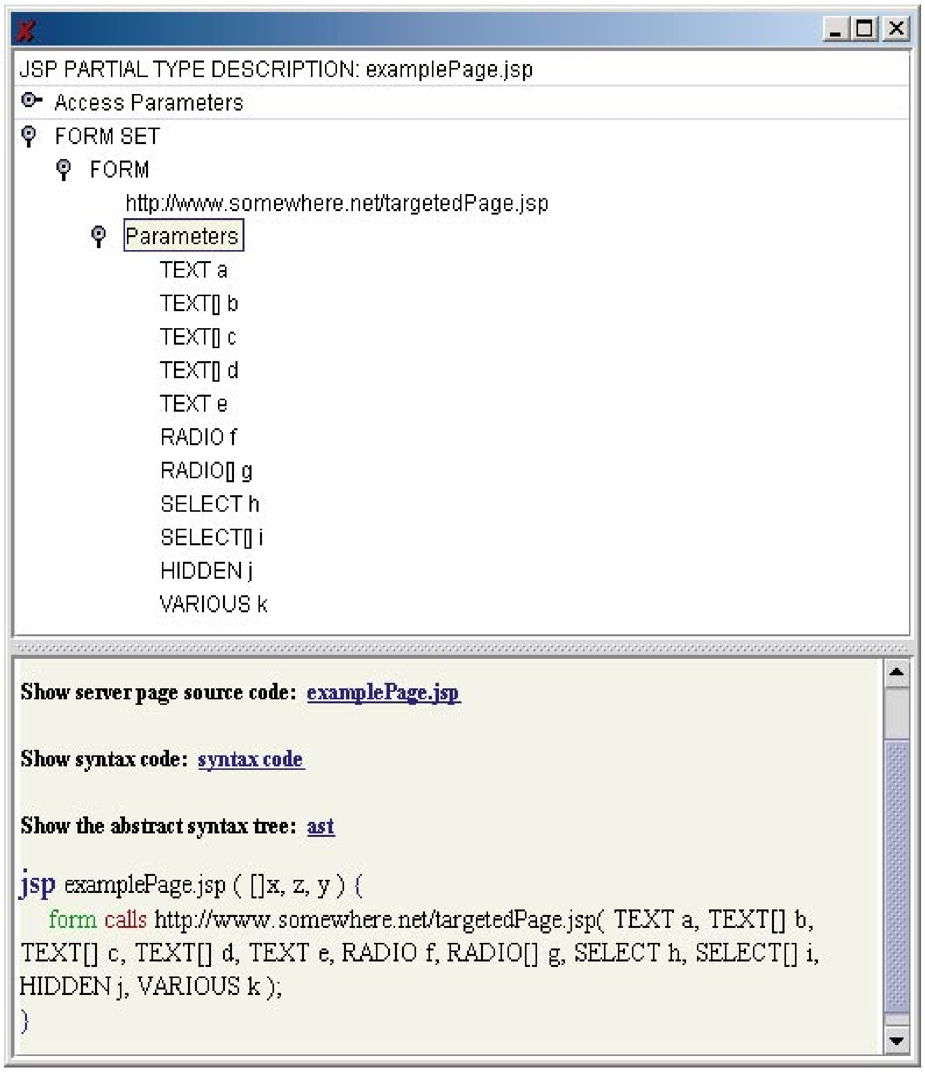
abstract syntax trees, type information, warnings and the linkage structure of the system in quest under several different viewpoints.

Listing 6.1

```
<html>
  <head><title>Example Page</title></head>
  <body>
    <form action="http://www.somewhere.net/targetedPage.jsp" method="get">
       <input type="text" name="a">
       <input type="text" name="b">
       <input type="text" name="b"><%
       for (i=0;i<2;i++){%> <input type="text" name="c"><%}
       if (cond){%> <input type="text" name="d"> <%}
       if (cond){%> <input type="text" name="e"> <%}
         else   { if (cond){%> <input type="text" name="e"> <%}
                    else   {%> <input type="text" name="e"> <%}
                }%>
       <input type="radio" name="f">
       <input type="radio" name="f"><%
       for (i=0;i<2;i++){%> <input type="radio" name="f"><%}
       if (cond){%> <input type="radio" name="g"> <%}%>
       <select name="h"><option>A<option>B</select>
       <select multiple name="i"><option>A<option>B</select>
       <input type="hidden" name="j"><%
       if (cond){%>
         <input type="radio" name="k">
         <input type="radio" name="k"><%
       } else {%>
         <select name="k"><option>A<option>B</select><%
       }%>
    </form><%
    v1=request.getParameterValues("x");
    v2=request.getParameter("y");
    v3=request.getParameterValues("z");
    if (cond) { v4=request.getParameter("z"); }
    %>
  </body>
</html>
```

Ensuring Robust Block Structure

As a basic feature JSPick encounters violations of the combined Java/XHTML block structure: Java and markup language block structure should be compatible as described in section 3.6.



JSP PARTIAL TYPE DESCRIPTION: examplePage.jsp

Access Parameters

FORM SET

FORM

http://www.somewhere.net/targetedPage.jsp

Parameters

TEXT a
TEXT[] b
TEXT[] c
TEXT[] d
TEXT e
RADIO f
RADIO[] g
SELECT h
SELECT[] i
HIDDEN j
VARIOUS k

Show server page source code: examplePage.jsp

Show syntax code: syntax code

Show the abstract syntax tree: ast

```
jsp examplePage.jsp ( []x, z, y ) {
    form calls http://www.somewhere.net/targetedPage.jsp( TEXT a, TEXT[] b,
TEXT[] c, TEXT[] d, TEXT e, RADIO f, RADIO[] g, SELECT h, SELECT[] i,
HIDDEN j, VARIOUS k );
}
```

Figure 6.1: JSPick Screenshot. The figure shows a partial type description window. Such a windows displays type information about the web signature of a Java Server Page and the form types of its possibly several contained forms.

Recovering Form Types

For every form information about the dynamically provided form input capabilities is created. The information inferred with respect to a given form is called its form type in the sequel. Consider the JSP example page in listing 6.1. One kind of window generated by JSPick is shown in Figure 6.1 which prepares the inferred information in several manners. For a form each targeted formal parameter is given[2]. If it is sure that always exactly one control is generated for a certain parameter then it is given as a single parameter. Otherwise it is given as an array parameter. Thereby a collection of radio buttons is one single control: a set of radio buttons offers the user one capability to select one single data item and is therefore conceptually equal to one single select menu. A multiple select menu is considered a collection of controls. A multiple select menu is conceptually equal to a set of check boxes: each of its options offers the user a capability to select or deselect a data item independently from the other menu options. The conceptual view of radio buttons and multiple select menus is summarized again in the informal equation 6.1.

$$\begin{array}{rc} & \texttt{mulitple radio buttons} \\ \equiv & \texttt{one single select menu} \\ \not\equiv & \texttt{one multiple select menu} \\ \equiv & \texttt{multiple check boxes} \end{array} \tag{6.1}$$

As a further information the control kind of each targeted formal parameter is given. If it is possible, that a parameter is targeted by different kinds of controls it is given the various-control, which is a pseudo control that has been introduced merely for this purpose. The most succinct JSPick presentation of the web signature and the form type that is extracted from the code in listing 6.1 is the following:

```
jsp examplePage.jsp( []x, z, y){
  form calls http://www.somewhere.net/targetedPage.jsp(
     TEXT a, TEXT[] b, TEXT[] c, TEXT[] d, TEXT e, RADIO f,
     RADIO[] g, SELECT h, SELECT[] i, HIDDEN j, VARIOUS k
  );
}
```

Summing up, a form type maps each formal parameter targeted by the form to a type consisting of a control kind and a possibly array annotation. Thereby the typing is oriented towards the NSP parameter rules given in section 3.3. The presented form types already provide the developer with a valuable debugging information. The second kind of information are extracted page signatures.

[2]We adopt NSP terminology in the context of Java Server Pages: a formal parameter is a name that is requested in a server page. A targeted formal parameter is a control's name-attribute. A formal parameter targeted by a form is the name-attribute of a possibly generated control contained in the form.

Recovering Page Signatures

The JSPick page signatures are crucially motivated by an insight into the specific interplay between HTML/XHTML forms and JSP server pages: the request.getParameter-method of the request-object should only be used for a parameter if it is sure that the parameter has at most one value[3]. Formulated in another way, if a parameter might have more than one value the getParameterValues-method should be used. This guideline motivates the way JSPick infers the web signature. If in a page a parameter is only requested by a getParameterValues-method, it is a formal array parameter. If in a page a parameter is possibly requested by a getParameter-method, it is a formal single parameter. Based on inferred form types and web signatures, JSPick can detect potential violations of the guideline described above and can generate an appropriate warning.

Implementation

The implementation of the JSPick reverse engineering tool encompasses 4,5 klocs. It is based on both standard [176] and innovative [141] compiler construction techniques. The state-of-the-art compiler generator ANTLR [159][142] has been employed. A detailed discussion of features, design, and implementation of JSPick is provided by [127].

Further Work

The information extracted by JSPick can be analyzed with respect to several classes of potential sources of error. For example, it is easy to implement reports on the following non-mutual exclusive indicators for a flawed design:

- A parameter is requested by a getParameter-method, but it is not provided by a targeting form.
- A parameter is provided by a form, but it is not requested in the targeted Java Server Page anywhere.
- Forms targeting the same Java Server Page may target different formal parameters.
- A form targets a non-existing form.
- A parameter may be requested by both a getParameter-method and a getParameterValues-method.
- A formal parameter that is targeted by a password control may be targeted by another non-password control.

[3]This is clearly stated in the online API documentation of the Java Servlet technology 2.2. The reason for this is obvious: the getParameter-method only returns the first value if it is applied to a string parameter [49]. More seriously in the elder Servlet [48]versions 2.1a this behavior is just proposed and the return value has been implementation dependent in such cases.

6.2 Formal Semantics of JSPick

This chapter formalizes the semantics of the core functionality of the reverse engineering tool JSPick, i.e. the recovery of a high-level description of the form parameters. The semantics is formalized as a pseudo-evaluation. A pseudo-evaluation runs a program with non-standard values instead of concrete values for the purpose of static program analysis. The pseudo-evaluation technique has been introduced with type checking in the GIER ALGOL compiler [129][128], where a program is executed on types instead of values[4]. Pseudo-evaluation has been taken up with other, more sophisticated notions of program analysis like data flow analysis [113] or abstract interpretation [43], each with other emphasis.

In JSPick pseudo-evaluation is preceded by parsing a program with respect to an non-standard abstract syntax, called pseudo-syntax in the sequel, that consists only of constructs that are relevant with respect to the desired program analysis. The pseudo-evaluation is formalized as a semantics function, i.e. in denotational style[5]. The complete definition of the semantics function is given in Figure 6.2 for easy reference.

First the JSPick pseudo-syntax is given by an extended context free grammar[6]. It picks up and formalizes the narrowed viewpoint of server pages syntax that has been necessary to describe the NSP coding rules as explained in section 3.3. Consequently the JSPick pseudo-syntax consists of controls as basic building blocks plus sequencing, if-structure, if-else-structure, switch structures, and loop. JSPick does not distinguish between completed and uncompleted switch structures in the sense of NSP for the following reason: in NSP it is a coding convention, that all branches of a switch structure must be ended by a break-statement. But in JSPick this cannot be demanded, because JSPick is designed as a tool for existing code. Fortunately it does not pose a problem, a useful form type inference can be defined with respect to another switch structure distinction. JSPick distinguishes between unique and arbitrary switch structures. In a unique switch structure all branches end with a break statement and the last branch is a default branch. All other switch-structures are arbitrary switch structures.

[4]Another early usage of the pseudo-evaluation technique is object code optimization [100] in ALGOL compilers.

[5]The original paper on pseudo-evaluation [129] gives an operational-style specification with respect to stack transformations.

[6]The production rules of an extended context free grammar may have regular expressions as right hand sides. Nonterminals are underlined. The syntactic category that corresponds to a given nonterminal is depicted in bold face.

$$
\begin{array}{rcll}
\underline{\texttt{pseudo}} & \texttt{::=} & & (\underline{\texttt{element}})\texttt{+} \\[1em]
\underline{\texttt{element}} & \texttt{::=} & & \underline{\texttt{control}} \\
 & & | & \texttt{if}\ \underline{\texttt{pseudo}} \\
 & & | & \texttt{ifElse}\ \underline{\texttt{pseudo}}\ \underline{\texttt{pseudo}} \\
 & & | & \texttt{switch}^{\texttt{arbitrary}}\ (\underline{\texttt{pseudo}})\texttt{+} \\
 & & | & \texttt{switch}^{\texttt{unique}}\ (\underline{\texttt{pseudo}})\texttt{+} \\
 & & | & \texttt{loop}\ \underline{\texttt{pseudo}}
\end{array}
$$

The usual types of HTML/XHTML controls are supported. Every control node carries the information about the name of the targeted formal parameter. The set of names is not specified.

$$
\begin{array}{rcll}
\underline{\texttt{control}} & \texttt{::=} & & \underline{\texttt{controltype}}\ \underline{\texttt{name}} \\[1em]
\underline{\texttt{controltype}} & \texttt{::=} & & \texttt{text | textarea | password} \\
 & & | & \texttt{hidden | checkbox | radiobutton} \\
 & & | & \texttt{singleselect | multipleselect} \\[1em]
\underline{\texttt{name}} & \texttt{::=} & & n \in \boldsymbol{name}
\end{array}
$$

The types that are assigned to targeted formal parameters as part of a form type are given by a little context free grammar, too. A type can be either a basic type or a basic type together with an array annotation. The basic types differ from the control types of the pseudo-syntax. It is not distinguished between single select menus and multiple select menus. The various pseudo-control is introduced.

$$
\begin{array}{rcll}
\underline{\texttt{parametertype}} & \texttt{::=} & & \underline{\texttt{basictype}}\ \texttt{|}\ \underline{\texttt{basictype}}\ [\,] \\[1em]
\underline{\texttt{basictype}} & \texttt{::=} & & \texttt{TEXT | TEXTAREA | PASSWORD} \\
 & & | & \texttt{HIDDEN | CHECKBOX | RADIOBUTTON} \\
 & & | & \texttt{SELECT} \\
 & & | & \texttt{VARIOUS}
\end{array}
$$

The semantics of JSPick form type inference is specified as a semantics function (6.2) that maps a pseudo-syntax tree to a form type. A form type is a finite partial function that assigns types to parameter names. Initially the semantics function is applied to the entire content of a form.

$$[\![\_]\!] : \boldsymbol{pseudo} \rightarrow \boldsymbol{name} \mathrel{+\!\!\!\!+\!\!\!\!\rightarrow} \boldsymbol{parametertype} \tag{6.2}$$

In order to define the semantics function it is helpful to have an auxiliary function (6.3) at hand that yields the contained basic type for every parameter type.

$$\Downarrow \_ \; : \boldsymbol{parametertype} \rightarrow \boldsymbol{basictype}$$

$$\Downarrow t = \begin{cases} t & , \; t \in \boldsymbol{basictype} \\ t' & , \; t = t'[\,] \end{cases} \tag{6.3}$$

The form type of a single control is defined only for the formal parameter that is targeted by the control. If the control is a multiple select menu the parameter is assigned the select basic type and an array annotation. If it is a single select menu the parameter is assigned the select basic type only. In all other cases simply the control type is assigned to the parameter, without array annotation.

$$[\![\mathit{controltype}\ \mathit{name}]\!] = \lambda n\,.\begin{cases} \bot & , n \neq \mathit{name} \\ \mathtt{TEXT} & , \mathit{controltype} = \mathtt{text} \\ \mathtt{TEXTAREA} & , \mathit{controltype} = \mathtt{textarea} \\ \mathtt{PASSWORD} & , \mathit{controltype} = \mathtt{password} \\ \mathtt{HIDDEN} & , \mathit{controltype} = \mathtt{hidden} \\ \mathtt{CHECKBOX} & , \mathit{controltype} = \mathtt{checkbox} \\ \mathtt{RADIOBUTTON} & , \mathit{controltype} = \mathtt{radiobutton} \\ \mathtt{SELECT} & , \mathit{controltype} = \mathtt{select} \\ \mathtt{SELECT}[\,] & , \mathit{controltype} = \mathtt{multipleselect} \end{cases} \tag{6.4}$$

The if-construct is semantically equivalent to an arbitrary switch structure and the if-else-construct is semantically equivalent to an unique switch structure in the way defined in the equations (6.5).

$$\begin{aligned} [\![\mathtt{if}\ p\ ]\!] &= [\![\mathtt{switch}^{\mathtt{arbitrary}}\ p\ ]\!] \\ [\![\mathtt{ifElse}\ p_1\ p_2\ ]\!] &= [\![\mathtt{switch}^{\mathtt{unique}}\ p_1\ p_2\ ]\!] \end{aligned} \tag{6.5}$$

Equation 6.6 defines how a form type is assigned to a unique switch structure[7]. For a unique switch structure it is ensured, that exactly one branch is executed. This fact can be exploited to possibly infer a single parameter type for a targeted formal parameter. Assume an arbitrary fixed name. If none of the branches yields a control that targets that name, the switch structure does not, too. If all branches always target that name with exactly one control the complete switch does so, too. If at least one branch targets the name but it is not sure that it produces exactly one control the switch targets the name with a control array. Equally if at least one branch targets the name and at least one branch does not target the name the switch targets the name with a control array. Thereby if all branches target the name with the same kind of control, the switch targets the name with this uniquely known control, otherwise it targets

[7]Free occurrences of the meta type variable t are implicitly existentially quantified in the equations of this section.

the name with the special various control.

$$[\![\texttt{switch}^{\texttt{unique}} \quad p_1 \ldots p_n]\!] =$$
$$\lambda\, n\,.\begin{cases} \bot & , \underset{1\le i\le n}{\forall}([\![p_i]\!]n)\uparrow \\ t & , \underset{1\le i\le n}{\forall}([\![p_i]\!]n) = t \in \boldsymbol{basictype} \\ \texttt{VARIOUS} & , \underset{1\le i\le n}{\forall}([\![p_i]\!]n) \in \boldsymbol{basictype} \\ t[\,] & , \underset{1\le i\le n}{\forall}\big(([\![p_i]\!]n)\uparrow \ \vee\ \Downarrow([\![p_i]\!]n) = t \in \boldsymbol{basictype}\big) \\ \texttt{VARIOUS}[\,] & , else \end{cases} \tag{6.6}$$

For an arbitrary switch structure it is not sure, that exactly one of the branches is executed. Therefore the switch structure either does not target a given name or targets that name with a control array. Apart from that arbitrary switches are equal to unique switches with regard to the form type. Therefore equation 6.7 immediately arises form equation 6.6 by dropping the second and third line.

$$[\![\texttt{switch}^{\texttt{arbitrary}} \quad p_1 \ldots p_n]\!] =$$
$$\lambda\, n\,.\begin{cases} \bot & , \underset{1\le i\le n}{\forall}([\![p_i]\!]n)\uparrow \\ t[\,] & , \underset{1\le i\le n}{\forall}\big(([\![p_i]\!]n)\uparrow \ \vee\ \Downarrow([\![p_i]\!]n) = t \in \boldsymbol{basictype}\big) \\ \texttt{VARIOUS}[\,] & , else \end{cases} \tag{6.7}$$

A loop targets every formal parameter that is targeted by its body with a control array.

$$[\![\texttt{loop}\, p\ ]\!] = \lambda\, n\,.\begin{cases} \bot & , ([\![p]\!]n)\uparrow \\ \big(\Downarrow([\![p]\!]n)\big)[\,] & , else \end{cases} \tag{6.8}$$

The form type of sequences of document parts is defined in equation 6.9.

$$[\![p_1 \ldots p_n]\!] =$$
$$\lambda\, n\,.\begin{cases} \bot & , \underset{1\le i\le n}{\forall}([\![p_i]\!]n)\uparrow \\ [\![p_i]\!]n & , \underset{1\le i\le n}{\exists!}([\![p_i]\!]n)\downarrow \\ \texttt{radiobutton} & , \begin{array}{ll} & \underset{1\le i\le n}{\forall}\big(([\![p_i]\!]n)\uparrow \ \vee\ \Downarrow([\![p_i]\!]n) = \texttt{radiobutton}\big) \\ \wedge & \underset{1\le i\le n}{\exists}\big(([\![p_i]\!]n) = \texttt{radiobutton}\big)\end{array} \\ t[\,] & , \underset{1\le i\le n}{\forall}\big(([\![p_i]\!]n)\uparrow \ \vee\ \Downarrow([\![p_i]\!]n) = t \in \boldsymbol{basictype}\big) \\ \texttt{VARIOUS}[\,] & , else \end{cases} \tag{6.9}$$

Assume a sequence document parts and that at least one of the parts targets a given name. If there is only one part that targets the name and furthermore the part targets the name with a single control then it is safe that the sequence targets the name with this single control. If all parts that target the name provide the same kind of control other than radio button, then the sequence targets the name with an array of the given control. Radio buttons are special, because they together provide a control. If a name is assigned a radio button and an array annotation this means only that it is not sure whether a radio button control is generated, it cannot mean that possibly more than one control is generated. Therefore if all document parts that target a given name provide radio buttons and at least one part is safe to provide a single radio button control, the sequence targets the name with a single radio button control[8]. At a last rule, if several parts target a given name with different kinds of control, the sequence targets the name with an array of various controls.

[8]JSPicks treats the generation of a one single radio button as a correct alternative, too. This treatment of radio buttons once more points up that JSPick is a tool for existing code: a lot of web designer deliberately use a single radio button this way as a kind of check box, though this is not in accordance with recommended good practice like e.g. [125].

$$[\![\mathit{controltype}\ \mathit{name}]\!] = \lambda n\,.\begin{cases} \bot & , n \neq \mathit{name} \\ \texttt{TEXT} & , \mathit{controltype} = \texttt{text} \\ \texttt{TEXTAREA} & , \mathit{controltype} = \texttt{textarea} \\ \texttt{PASSWORD} & , \mathit{controltype} = \texttt{password} \\ \texttt{HIDDEN} & , \mathit{controltype} = \texttt{hidden} \\ \texttt{CHECKBOX} & , \mathit{controltype} = \texttt{checkbox} \\ \texttt{RADIOBUTTON} & , \mathit{controltype} = \texttt{radiobutton} \\ \texttt{SELECT} & , \mathit{controltype} = \texttt{select} \\ \texttt{SELECT}[\,] & , \mathit{controltype} = \texttt{multipleselect} \end{cases}$$

$$[\![\texttt{if}\ p]\!] = [\![\texttt{switch}^{\texttt{arbitrary}}\ p]\!]$$
$$[\![\texttt{ifElse}\ p_1\ p_2]\!] = [\![\texttt{switch}^{\texttt{unique}}\ p_1\ p_2]\!]$$

$$[\![\texttt{switch}^{\texttt{arbitrary}}\ \ p_1 \ldots p_n]\!] = \lambda n\,.\begin{cases} \bot & , \underset{1\leq i\leq n}{\forall} ([\![p_i]\!]n)\uparrow \\ t[\,] & , \underset{1\leq i\leq n}{\forall} (([\![p_i]\!]n)\uparrow\ \vee\ \Downarrow([\![p_i]\!]n) = t \in \boldsymbol{basictype}) \\ \texttt{VARIOUS}[\,] & , \mathit{else} \end{cases}$$

$$[\![\texttt{switch}^{\texttt{unique}}\ \ p_1 \ldots p_n]\!] = \lambda n\,.\begin{cases} \bot & , \underset{1\leq i\leq n}{\forall} ([\![p_i]\!]n)\uparrow \\ t & , \underset{1\leq i\leq n}{\forall} ([\![p_i]\!]n) = t \in \boldsymbol{basictype} \\ \texttt{VARIOUS} & , \underset{1\leq i\leq n}{\forall} ([\![p_i]\!]n) \in \boldsymbol{basictype} \\ t[\,] & , \underset{1\leq i\leq n}{\forall} (([\![p_i]\!]n)\uparrow\ \vee\ \Downarrow([\![p_i]\!]n) = t \in \boldsymbol{basictype}) \\ \texttt{VARIOUS}[\,] & , \mathit{else} \end{cases}$$

$$[\![\texttt{loop}\ p]\!] = \lambda n\,.\begin{cases} \bot & , ([\![p]\!]n)\uparrow \\ \big(\Downarrow([\![p]\!]n)\big)[\,] & , \mathit{else} \end{cases}$$

$$[\![p_1 \ldots p_n]\!] = \lambda n\,.\begin{cases} \bot & , \underset{1\leq i\leq n}{\forall} ([\![p_i]\!]n)\uparrow \\ [\![p_i]\!]n & , \underset{1\leq i\leq n}{\exists\,!} ([\![p_i]\!]n)\downarrow \\ \texttt{radiobutton} & , \underset{1\leq i\leq n}{\forall} (([\![p_i]\!]n)\uparrow\ \vee\ \Downarrow([\![p_i]\!]n) = \texttt{radiobutton}) \\ & \quad \wedge \underset{1\leq i\leq n}{\exists} ([\![p_i]\!]n) = \texttt{radiobutton} \\ t[\,] & , \underset{1\leq i\leq n}{\forall} (([\![p_i]\!]n)\uparrow\ \vee\ \Downarrow([\![p_i]\!]n) = t \in \boldsymbol{basictype}) \\ \texttt{VARIOUS}[\,] & , \mathit{else} \end{cases}$$

Figure 6.2: JSPick Pseudo-Evaluation. The figure contains the complete specification of the semantics of the reverse engineering tool JSPick with respect to type inference of form types. JSPick is a design recovery tool for Java Server Pages based presentation layers.