Typed Server Pages

Am Fachbereich Mathematik und Informatik
der Freien Universität Berlin
eingereichte Dissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften

vorgelegt von Dirk Draheim

Berlin, 2002
Tag der Disputation: 29.November 2002

Gutachter: Prof. Dr. Elfriede Fehr
PD Dr. Martin Große-Rhode

Abstract

In this dissertation a statically typed server pages approach is elaborated. The resulting notions of correctness ensure the type-safe interplay of dynamically generated web forms and targeted software components on the one hand side and the generation of documents that are valid with respect to a given user interface description language on the other hand. The approach contributes the notion of tag support for gathering complex user defined data, the notion of exchanging complex objects virtually across the web user agent, the notion of typed server-side calls to server pages, and the notion of higher order server pages.

Web applications are ubiquitous. Ultra-thin client based tiered enterprise applications become more and more important. Due to this a considerable number of innovative web technologies has been provided recently. Currently web applications and web based system architecture are fields of intensive research activity. Server pages technology is a state-of-the-art in this field. The contributions of this work target stability, maintainability and reusability of server pages based systems. The findings are programming language independent, a concrete programming language can be amalgamated with the found concepts in a way that is conservative with respect to the language's semantics.

In this dissertation a server pages based presentation layer is characterized abstractly as a closed collection of typed dialogue methods. Based on this coding guidelines and rules are defined that together informally provide the desired notions of type correctness and description correctness. The static semantics of the new server pages approach is defined as a Per Martin-Löf style type system with respect to an amalgamation with a minimal imperative programming language and a sufficiently complex equi-recursive type system. The "model two architecture" of web based system applications is analyzed in order to provide a further justification of the proposed approach. NSP/Java, a concrete amalgamation of the concepts with the programming language Java, is discussed. An operational semantics of the resulting technology is described as a transformation of NSP/Java technology to Java Server Pages technology. Furthermore JSPick is presented, a reverse engineering tool for Java Server Pages based presentation layers. A formal semantics of this CASE tool is given as pseudo-evaluation.

The server pages approach is part of a proposed holistic approach to modeling and developing form based, submit/response style systems. A case study exemplifies how the impedance mismatch between modeling and implementation is mitigated by the approach.

Contents

List of Figures

List of Code Examples

Chapter 1

Introduction

In this dissertation a strongly typed server pages technology is proposed. Server pages technology is a state-of-the-art in the field of web technology.

Web applications are ubiquitous. For every company that wants to stay competitive it is not the question whether to deploy internet technology, but how to deploy it [146]. Highly skilled workers are needed to operate e-commerce technology [183], indeed a research study [77] conducted by the Gartner Group has shown that more than three-quarter of the cost of building an e-commerce site is labor related. Consequently web technology is worth looking at.

The viewpoint of this dissertation is a software architect's and a theoretical computer scientist's viewpoint. Ultra-thin client based tiered enterprise applications benefit from scalability and maintainability. Server pages technologies are widely used in the implementation of ultra-thin client applications. Unfortunately the low-level CGI programming model shines through in these technologies, especially user data is gathered in a completely untyped manner. In this work a strongly typed server pages technology is designed from scratch. The contributions target stability and reusability of server pages based systems. The findings are programming language independent. The results are formalized. The following concepts are combined:

- Parameterized server pages. A server page possesses a specified signature that consists of formal parameters, which are native typed with respect to a type system of a high-level programming language.

- Support for complex types in writing forms. New structured tags are offered for gathering arrays and objects of user defined types.

- Exchanging objects across the web user agent. Server side programmed objects may be actual form parameters and therefore passed to client pages and back, either as messages or virtually as objects.

- Higher order server pages. Server pages may be actual form parameters.

- Statically ensured client page type safety. The type correct interplay of dynamically generated forms and targeted server pages is checked at compile-time.

- Statically ensured client page description safety. It is checked at compile-time if generated page descriptions always are valid with respect to a defined client page description language.

- No unresolved links. It is statically ensured that all generated forms and links point to existing targets within the system dialogue.

- Active controls. NSP direct input controls are dynamically type safe, that is dynamic type checks with respect to data entered by the user are statically ensured. Equally checks with respect to required data are statically ensured.

The proposed server pages technology NSP (Next Server Pages) does not only overcome drawbacks of current CGI based technologies and helps the developer immediately to code cleaner, better reusable, more stable systems. The NSP approach is oriented towards improved web application architecture and improved development techniques from the outset:

- Enabling technology for improved web-based application architecture and design. In NSP a server-side call to a server page is designed as a parameter-passing procedure call, too. This enables functional decomposition of server pages and therefore helps decoupling architectural issues and implementing design patterns.

- Reverse engineering. The NSP concepts are exploited by the fully implemented reverse engineering tool JSPick, which recovers web signatures and form types from Java Server Pages based presentation layers. A formal semantics of the tool is given in pseudo-evaluation style.

- Seamless integration with form-oriented analysis. There exist canonical mappings from form charts, which model systems as constraint language annotated bipartite state transition diagrams, to NSP based systems.

- Formal semantic basis for type safety. The core type system of NSP is given as a convenient Per Martin-Löf style type system. This enables precise reasoning about the NSP concepts.

The aforementioned concepts target to overcome certain drawbacks of CGI (common gateway interface) based technologies, which today provide the single most important means to build presentation layers of web-based systems, as explained in the sequel. A web-based system consists of a set of server side scripts. A script is a code unit that is called across the net by the user by submitting a form or selecting a link. On behalf of this the script triggers business logic and eventually produces a client page that is sent back to the user. The client

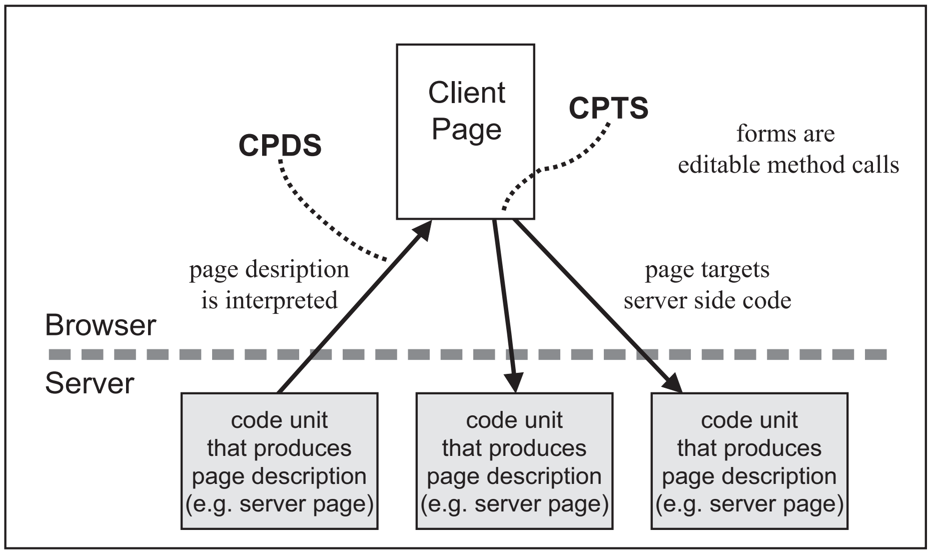

Client Page
CPDS
CPTS
forms are editable method calls
page desription is interpreted
page targets server side code
Browser
Server
code unit that produces page description (e.g. server page)
code unit that produces page description (e.g. server page)
code unit that produces page description (e.g. server page)

Figure 1.1: CPDS and CPTS. The figure shows the interplay of server side scripts and the browser and visualizes the notions of client page description safety (CPDS) and client page type safety (CPTS).

page is coded in the client page description language HTML or XHTML. The client page description is interpreted by a browser (Fig. 1.1). A page presents the user information and offers her one or more forms and links. Every form and every link targets one of the server-side scripts, that is the user gets explicit control over the dialogue flow. For the NSP approach it is crucial, that a form may be viewed as an editable method call, gathering actual parameters from the user. A form may have pre-selected actual parameters, hidden parameters in more technical terms. It follows that links can be subsumed conceptually under forms.

We coin the two terms *client page description safety* and *client page type safety* for two desired properties of web applications.

Definition 1.0.1 (Client page description safety CPDS) *A web application has the property of client page description safety if its server scripts always only produce valid page descriptions with respect to a defined client page description language.*

Definition 1.0.2 (Client page type safety CPTS) *A web application has the property of client page type safety if the client pages generated by the server scripts always only contain such forms and links, that provide exactly such actual parameters that are expected by the respective targeted server scripts.*

We use terms like client page type error, correctness, or checking according to definitions 1.0.1 and 1.0.2. We have chosen to use some more general, less technical terms than might be actually needed at first sight like client page description language instead of HTML/XHTML[1]. We did so in order to foster the viewpoint, that web applications are instances of a more general class of ubiquitous applications, which may be characterized as submit/response style applications. Furthermore we wanted to emphasize that the concepts that are investigated in the context of NSP do not stuck to concrete web technology, but could add value to every future ultra-thin client architecture supporting technology.

Current CGI based server-side scripts receive values submitted by a form or link conceptually as stream of named values. In raw CGI programming, the name/value pairs must be retrieved from an input parameter string. The several well known scripting languages and technologies, like PHP [108], Perl [174], Tcl [137], Active Server Pages [110] [147], Python Server Pages [4], Java Servlets [49], and Java Server Pages [143][20] support the retrieval of submitted values with appropriate data structures. But beyond this, these technologies don't support any appropriate notion of a server script signature, i.e. the interesting types of the received string-valued parameters with respect to a business logic or, in more code-oriented terms, with respect to a high-level programming language type system are not supported.

In contrast, NSP offers appropriate tags to define a complex server page signature. Based on this the definition 1.0.2 of client page type safety obtains a precise meaning in the context of NSP, because parameters that are expected by a server page are just the parameters specified by the server page's signature. NSP provides both client page description checking and client page type checking. From the viewpoint of an NSP type system the generated client pages are the actual code, which has to be considered. The generated code is naturally not available at deployment time, therefore NSP defines guidelines and rules for writing the server pages which are

- non-prohibitive: all reasonable applications of scripting are still allowed.
- sufficient: client page description and type safety are ensured.
- convenient: the coding guidelines and rules target the NSP developer. They are natural and easy to understand. The coding guidelines and rules provide the informal definition of the NSP type system.

Conservative Amalgamation

The NSP concepts are programming language independent results. They are reusable. They must be amalgamated with a concrete programming language. For every such amalgamation a concrete non-trivial language mapping must be

[1]Actually NSP guarantees only a weaker client page description safety than valid HTML/XHTML output safety as explained in section 3.6, but this is due to pragmatic reasons and not due to technical reasons: furthermore the NSP notion of client page description safety is justified by existing browser technology.

provided. The NSP concepts are designed in such a way that concrete amalgamations are conservative with respect to the programming language. That is the semantics of the programming language and especially its type system remain unchanged in the resulting technology. In this dissertation the NSP concepts are explained through a concrete amalgamation with the programming language Java. As a result of conservative amalgamation the NSP approach does not restrict the potentials of JSP in any way, for example its state handling facility, the Servlet API session concept, is available as a matter of course. Formal semantics of an NSP core type system is given with respect to an amalgamation with a minimal imperative programming language.

Integration with Form Orientation

NSP based systems benefit from type safety. But beyond this NSP's capability for server pages functional decomposition suggests to reconsider web application architecture and design. In the NSP approach a server pages based presentation layer is characterized as a closed collection of typed dialogue methods, an abstract viewpoint that is shared by form-oriented analysis [178]. Based on functional decomposition it is possible to find canonical mappings from form charts [60] to NSP technology. Consequently NSP becomes an integral part of Form Orientation, a holistic approach for modeling and developing a certain kind of widespread form-based, submit/response style enterprise application, which contributes form storyboarding, form chart modeling, a dialogue constraint language, layered data modeling and the NSP technology.

Related Work

Detailed discussions of related work are provided throughout the paper. Furthermore a comprehensive appraisal of related work is given in section 8.1. To our knowledge the NSP project is the only approach that investigates client page description safety and client page type safety with respect to server pages technology. Thereby the results are programming language independent. The contributions are summarized in section 8.3.

There are two projects on imperative domain specific languages [53] for web programming, i.e. MAWL [8] and Bigwig [18] discussed in section 8.1.2. Several projects address active web publishing in the context of functional programming discussed in section 8.1.6.

It is necessary to mention the debatable important opinion [138] known as Ousterhout's dichotomy. A division is made between system programming languages for programming components from scratch and scripting languages for programming component glue. It is argued that the typeless nature of a scripting language is crucial - a property that enables rapid development. The objectives of the present work contradict this core argument of Ousterhout's dichotomy.

Dissertation Outline

The dissertation is organized as follows. Chapter 2 motivates the NSP approach and explains the overall structure of an NSP document as well as the NSP interaction controls, thereby introducing the concepts of active direct input control and active single select control. Chapter 3 explains the NSP support for gathering array data and data of user defined type. The NSP coding guidelines and rules are given, which together provide the informal description of the NSP type system. The NSP type system is specified in chapter 7 as a Per Martin-Löf style type system for Core NSP, which is the amalgamation of NSP concepts with a minimal imperative programming language and a sufficiently complex type system for modeling all relevant aspects of a modern high-level programming language's type system. Example type derivations for some Core NSP server pages are provided in appendix B. Furthermore this appendix summarizes abstract syntax and typing rules of the NSP definition given in chapter 7, whereas in appendix C an XML DTD for the Core NSP language and an SGML DTD for the Core NSP user interface description language are given. In chapter 4 the architecture that underlies current web application frameworks is analyzed. The concept of server-side call to a server page is introduced as well as the concept of higher order server pages. Motivations of the concepts are given by explanations of their capability to improve web application architecture. In chapter 5 an operational semantics of the NSP/Java language amalgamation is given, i.e. a core reference implementation is defined by the description of a transformation to Java Server Pages and auxiliary technology. In chapter 6 the JSPick reverse engineering tool for design recovery of Java Server Pages based presentation layers is presented. Chapter 8 outlines further directions and provides a detailed discussion of related work. Appendix A is a case study, that exemplifies how to transform a given form chart system specification to NSP code.

Acknowledgements

Special thanks to my *spiritus rector* Prof.Dr.Elfriede Fehr for making possible this dissertation. Special thanks to PD Dr.Martin Große-Rhode for the appraisal of the dissertation. His comments on early versions of the text helped to improve the quality of the dissertation significantly.

Thanks to my *alma mater* "Freie Universität Berlin - Institut für Informatik" for extraordinary good labor conditions and especially thanks to the several institute members who lent a helping hand.

Thanks to Gerald Weber for his ideas and collaboration.

Chapter 2

Next Server Pages Preliminaries

The Next Server Pages technology (NSP) [57][62][127] is presented by the concrete amalgamation of NSP concepts with the programming language Java [79]. In the NSP approach a server page is considered to be code of a programming language with defined syntax and defined type system. In contrast, Java Server Pages (JSP) [143][20] code is just a convenient notation for Java Servlets [49]. It is essentially a mix of HTML and Java code, which may occur inside special opening respective closing scriptlet signs[1]. The semantics of a JSP server page is given by the effect of a preprocessor that yields a Java Servlet by placing the HTML parts of the server page into output statements and adding these statements to the Java code that occurs inside the scriptlet signs.

2.1 A Motivating Example

The improvements of the Servlet API over raw CGI [38] programming, support for retrieving values and a session mechanism, are available to JSP developers as a matter of course, anyway JSP technology has the same disadvantages as other CGI based technologies, as described in the introduction: JSP based systems may lack client page description and type safety. JSP does not offer a natural server-side call to server scripting code[2].

As an instructive example consider the system that comprises the two server pages given in listing 2.1 and listing 2.2. The first server page generates a registration client page that contains a form for gathering a customer's name

[1]Actually beyond being a convenient notation JSP technology offers some sophisticated mechanisms like tag libraries and JSP actions , especially such for supporting the integration of Java Beans into a server page. Furthermore JSP is intended to foster techniques for separating content from layout. But all these issues are orthogonal to the current discussion.

[2]The combination of the JSP actions `jsp:include` and `jsp:param` does not provide a usual type-safe parameter passing mechanism, it just provides a convenient notation for the low level Java Servlet include mechanism.

Listing 2.1

```
01 <html>
02   <head>
03     <title>Registration</title>
04   </head>
05   <body> <%
06     int j;
07     boolean c;
08     // computation of the variables j and c
09     for (int i=0; i<j; i++) { %>
10     <form action="http://localhost:8080/NewCustomer.jsp"> <%}%>
11       Name: <%
12       if (c) { %>
13            <input type="text" name="customer"><br> <%
14       } %>
15       Age: <input type="text" name="age"><br>
16       <input type="submit">
17     </form>
18   </body>
19 </html>
```

and age. This form targets a server page, that stores the received data to a database and forwards the request to another server page. Several dynamic errors may occur. First of all, the opening form tag in line 10 of listing 2.1 occurs inside a loop body. Only if the variable `j` has value `1` before the loop is entered, a valid HTML document is sent to the browser. The input capability for the customer name occurs underneath a control structure in line 13. Only if the boolean variable `c` evaluates to `true` the input capability will actually occur on the registration page. Listing 2.3 shows the result page for the case that variable `j` has value `3` and variable `c` evaluates to `false`. The fact that the form tag occurs three times in the client page is actually tolerated by some of the ubiquitous web browsers, nevertheless it must be considered a dynamic client page description error. For example, in the case that variable `j` has value `0` even a tolerant browser cannot recover from the error, because together with the form tag the vitally information about the targeted page is missing.

As an example for a dynamic client page type error, assume that the customer parameter of the business logic method in line 16 of listing 2.2 must not be a null object, i.e. the targeted server page relies on the reception of a customer name. However, it is not guaranteed that an input capability for the customer name is offered to the user by the registration page. If no such input capability occurs as in listing 2.3, no respective value is sent on submitting the form and consequently the effort to retrieve such a value in line 10 of listing 2.2 will result in yielding a null object.

Even more obviously, the developer would like to have a tool that gives a warning with respect to line 09 in listing 2.2, because a value bound to name

Listing 2.2

```
<html>
  <head>
    <title>NewCustomer</title>
  </head>
  <body> <%!
    import myBusinessModel.CustomerBase; %> <%
    String customer;
    int age;
    String foobar = request.getParameter("foobar");
    customer = request.getParameter("customer");
    try {
      age = Integer.parseInt(request.getParameter("age"));
    } catch(NumberFormatException e){
      // error handling
    }
    CustomerBase.createCustomer(customer,age);
    // further business logic %>
    <jsp:forward page="Somewhere.jsp"/>
  </body>
</html>
```

foobar will never be provided by a form of the registration page[3].

Another kind of dynamic type error may occur with respect to the customer age parameter. A user may enter a value that is not a number, which is an event that must be caught by the developer. Appropriate error handling code, either through client-side or server-side scripting, must be provided, which is a tedious and error prone task.

2.2 The NSP Document Structure

NSP modifies and augments XHTML [172] due to the special needs of developing parameterized strongly typed server pages. The modifications and augmentations are succinctly defined by an XML DTD of Core NSP in appendix C.1. Most importantly, an implementation of NSP, be it a full fledged container or a reference implementation that compiles NSP code to JSP code, will always only send XHTML to the browser, and consequently NSP technology can be used immediately with existing standard browsers, i.e. no plug-ins are needed[4].

[3]The JSPick tool presented in chapter 6 is able to detect such sources of error.

[4]Though NSP is an XML document that is interlaced with imperative code, its objectives must not be confused with that of Extensible Server Pages (XSP) [117], a dynamic XML technology. XSP is an integral part of the XML publishing framework Cocoon [39], which targets separation of concerns between content, logic, and style of web publishing applications. In contrast, the NSP approach is not oriented too much towards building information architectures [155] for the time being, but towards improving the stability of enterprise information

Listing 2.3

```
<html>
  <head>
    <title>Registration</title>
  </head>
  <body>
    <form action="http://localhost:8080/NewCustomer.jsp">
    <form action="http://localhost:8080/NewCustomer.jsp">
    <form action="http://localhost:8080/NewCustomer.jsp">
      Name:
      Age: <input type="text" name="age"><br>
      <input type="submit">
    </form>
  </body>
</html>
```

The name NSP/Java is used for the concrete language that results from merging NSP concepts with the programming language Java. If it is obvious from the context that not only the entirety of NSP technology concepts is meant the term NSP is used for NSP/Java, too. An NSP server page consists of a signature definition, a java definition block and a core document. There are NSP server pages that may be called across the net by a form or a link, others may be called by another NSP server page on the server side in order to be included. For NSP server pages that may be called across the net the core document consists of a head and a body, for the other kind of server page the core document content is enclosed in include tags. A first example is given in listing 2.4 and listing 2.5,which shows the NSP counterpart of the customer registration system, which has been discussed in section 2.1.

The signature of a server page is defined with appropriate parameter tags. Attributes are used for specifying the name and the type of a formal parameter.

Definition 2.2.1 (Web signature) *A web signature is a record type which consists of the several specified formal parameters of an NSP server page as labeled components.*

We coin another term for the concept of web signature, namely formal superparameter. Analogously, an actual superparameter is the entirety of actual parameters provided by a form or a link targeting an NSP server page with respect to a given formal superparameter.

Definition 2.2.2 (Web server page) *A web server page is an NSP server page that may be called across the net by a form or a link.*

Definition 2.2.3 (Include server page) *An include server page is an NSP server page that may be called for inclusion into another NSP server page.*

systems [102] with a tiered ultra-thin client architecture.

Listing 2.4

```
<nsp name="Registration">
  <html>
    <head>
      <title>Registration</title>
    </head>
    <body><java>
      boolean c;
      // computation of the variable c </java>
      <form callee="NewCustomer"><java>
        if (c) {</java>
          <input type="String" param="customer"></input><java>
        } else {</java>
          <hidden param="customer">"DefaultName"</hidden><java>
        } </java>
        Age: <input type="int" param="age"></input><br>
        <submit></submit>
      </form>
    </body>
  </html>
</nsp>
```

The web server pages are the front components of a tiered system's presentation layer, therefore we use the term dialogue method synonymously for such server pages throughout the dissertation, especially to distinguish them from business methods. Similarly we use the term dialogue submethod for NSP include server pages.

In NSP server pages Java code is be placed inside java tags. Some Java code may be placed between the signature definition and the server page core. It is called Java declaration in accordance with JSP terminology and hosts code that is intentionally independent from the single server page invocation, like e.g. import declarations or definition of state that is shared by all the several sessions. Java code that occurs in the server page core is executed upon invocation of the server page. The server page parameters are accessible in the inline Java code. In addition to the tags it is possible to use java expression tags as a controlled variant of direct, i.e. Java coded, writing to the output stream[5].

In NSP no special non-XML syntax for expression scripting elements like the JSP <%= and %> signs is available. Because of that it is not possible

[5]Note that a static type system like NSP's cannot prevent such dynamic errors that result from using the output stream to send computed description languages tags to the browser directly. Generally, using the output stream for sending description language is considered bad style and may lead to dynamic errors. It is an NSP rule that the output stream must not be used in a way that corrupts the otherwise type and description safe NSP system. For example, if a corrupted form that is caused by a prohibited use of the output stream contains invalid input capabilities or a wrong number of input capabilities, this is considered just as a dynamic error, like division by zero.

Listing 2.5

```
<nsp name="NewCustomer">
  <param name="customer" type="String"></param>
  <param name="age" type="int"></param>
  <java>import myBusinessModel.CustomerBase;</java>
  <html>
    <head>
      <title>NewCustomer</title>
    </head>
    <body>
      <java>
        CustomerBase.createCustomer(customer,age);
      </java>
      <forward callee="Somewhere"></forward>
    </body>
  </html>
</nsp>
```

to generate NSP tag parts, especially attribute values may not be generated. Element properties that may have to be provided dynamically are supported by elements instead of attributes in NSP. As a result an NSP server page is a completely block structured document and therefore it is possible to give a precise convenient definition for the syntax of NSP, which is the essential basis for detection and exploration of further concepts of syntax analysis and syntax manipulation. Moreover an NSP server page is a valid XML [19] document. Therefore NSP will benefit immediately from all new techniques developed in the context of XML. In particular, NSP can be used in combination with XML style sheet technologies [34][126].

NSP Forms and Hyperlinks

An NSP form specifies the targeted dialogue method via a callee attribute. Somewhat similar to the parameter passing mechanism in ADA [171], a targeted formal parameter of a called method is explicitly referenced by its name. For this purpose the tags in line 13 and 15 of listing 2.4 have parameter attributes.

In the present simple example the form provides actual parameters exactly for the formal method parameters, either by user input or hidden parameters. In general the overall notion of type-safe calls of NSP methods demands for sophisticated guidelines and rules for writing forms presented in chapter 3 and a new innovative widget set presented in section 2.3.

The syntax of NSP hyperlinks follows the NSP form syntax, listing 2.6 provides an example. Therefore the NSP hyperlink syntax is very natural, i.e. no low-level, tedious handling with special signs is needed in order to use hyperlinks with parameters. An NSP link element must not contain any other code than hidden controls and one linkbody element that gives the link text, in fact it must contain exactly one hidden control of correct type for every formal parameter of the targeted dialogue method.

Listing 2.6

```
<link callee="NewCustomer">
  <hidden param="customer">
    "John Q. Public"
  </hidden>
  <hidden param="age"> 32 </hidden>
  <linkbody> underlined link name </linkbody>
</link>
```

A Motivating Example

All the code of the lines 7 to 15 of listing 2.2, which is needed for the reception of parameters, becomes a two-line web signature declaration in listing 2.5.

In listing 2.1 the opening form tag is a loop body, which may lead to dynamic errors. In NSP this is prevented from the outset by NSP syntax. An element may only occur as a whole, i.e. with both opening and closing tag underneath a control structure.

It is not ensured that the registration page generated by listing 2.1 offers an input control for the customer name. In contrast, in the NSP code in listing 2.4 it is for example not allowed to omit line 13 without violating the NSP coding guidelines, i.e. without provoking an error message at compile time.

Obviously faulty requests to non-existent parameters like in line 9 of listing 2.2 are not possible in NSP code, because NSP is strongly typed and all formal server page parameters have to be declared.

Furthermore in an NSP system it is statically ensured, that data provided by the user is dynamically checked. That is, a server side dynamic error like the one described for the age parameter in listing 2.2, would not be possible in an NSP system because of the concept of active controls, which is introduced in section 2.3.1 and 2.3.3.

2.3 NSP Interaction Controls

NSP supports the usual HMTL/XHTML controls, but they are confined and elaborated further due to the special needs of a type-safe server pages technology. Each control is supported by its own element. Each control has a param attribute[6], which is used to directly specify the name of the targeted formal parameter. Another usage of the param attribute is specifying a field of a complex actual parameter of form message type as explained in section 3.5.

[6]The param-attribute replaces the name-attribute of HTML/XHTML controls.

2.3.1 NSP Active Direct Input Controls

In NSP the input tag is used for a direct input field only[7]. NSP provides different direct input fields for different programming language types. That is NSP is typed already on the level of controls[8]. The type attribute is used to specify the type of a direct input field[9]. The direct input controls of NSP are active input controls in the following sense. A form that contains a direct input field cannot be submitted if the user has entered data that is not type-correct with respect to the field's type. Instead of that an error message is presented to the user. Furthermore in NSP a formal parameter can be specified to be required by the developer. Then a form that contains a direct input field that targets this formal parameter cannot be submitted if the user has entered no data in this field. Again an appropriate error message is presented to the user. Dynamic type checking of data provided by the user and dynamically ensuring data entry are ubiquitous problems in web interface development. The developer must provide solutions by either client-side scripting with one of the ECMAScript [70][96] derivatives or server-side scripting, a tedious and error-prone task. With the NSP notion of active direct input controls it is possible to statically ensure the desired dynamic checks.

In NSP/Java the Java types int, Integer, String and the type Date of the JDBC API [186] are supported by direct input fields. If data entry for a formal parameter is optional and no data is entered by the user, a null object is passed as actual parameter on submit. The single exception to this rule is the type String: if no data is entered in a field for an optional parameter the empty string is sent instead of a null object. For formal int parameters data entry is implicitly required[10]. There are two further direct input capabilities for the type String, i.e. a text area and a password field.

Related Work

The XForms standard [69] allows for specifying constraints on data gathered by a form. Type correctness and entry requirements are important instances of possible constraints. But the XForms approach goes beyond these. It is possible to specify arbitrary calculations and validations on the gathered data. An event model allows for the appropriate reaction on the violation of the given constraints. Similarly the PowerForms technology [16] defines a declarative lan-

[7]In HTML/XHMTL the input tag is used for a couple of conceptually unrelated controls, like direct text input, radio buttons, check boxes etc. Every of these controls has its specifics, especially the behavior of non-direct input controls differs fundamentally from the behavior of direct input controls. Therefore in NSP every control type becomes a markup language element.

[8]In HMTL/XHTML controls always only yield text, i.e. pure string data.

[9]In HMTL/XHTML the type attribute is used to specify a widget kind.

[10]The type int is a Java primitive type and no null object is available for signalizing that no data has been entered. Alternatively it would be possible to select a certain int value to take over the role of the null object, most probably zero. But then the receiving dialogue method cannot distinguish between an actually entered data and the event, that no data has been entered.

guage for expressing input formats and interdependencies between input fields. Then client side code is generated for ensuring the declared constraints. Both technologies allow for sophisticated constraints, however the notion of NSP active controls is different. In XForms and PowerForms constraints can be given with respect to forms. But it is not ensured, that all forms targeting a critical server side action actually prevent the same errors from occurring. In contrast, in NSP, strictly based on the typing of server pages, the constraints are given for formal parameters of dialogue methods. This way it is possible to statically ensure the desired dynamic checks, that is NSP active direct input controls are dynamically type safe in the sense that they make NSP based systems type safe with respect to dynamically entered data[11].

2.3.2 Actual Object Parameters

In the NSP approach arbitrary objects can be passed virtually across the web, i.e. passed to the user agent and passed to a dialogue method on submit in the sequel. An object can be passed this way as hidden parameter, item of a select menu, or value of a radio button or check box[12]. A value of primitive type is just copied to the user agent and to the dialogue method on submit, thereby preserving its present type. But NSP supports passing of objects of arbitrary non-primitive types. Thereby, somewhat similar to the RMI parameter passing mechanism [164], two different parameter passing semantics are supported. First, as a default, a reference to the object in quest is passed to the receiving dialogue method. As an alternative, if the passed object is serializable [165], the object is copied and a reference to this copy is passed to the receiving dialogue method. Based on the semantics of serialization arbitrary deep copies of object nets can be passed as parameters. At the extreme, choosing to pass an object net as a completely deep copy fully coincides with the notion of message in the sense of [178], i.e. data dictionary objects.

A hidden parameter, select menu item, radio button or check box value must be given as a Java expression. This expression must have the type of the encompassing control. The encompassing tags implicitly switch to a Java expression modus, i.e. no additional expression tags are needed in order to explicitly signalize a Java expression. Furthermore this rule applies wherever appropriate in order to give a Java object as a certain desired property of a control. For example a default value may be given for direct input fields. For this purpose input field tags may contain a type-correct Java expression[13] [14]. Again the input field tags implicitly switch to a Java expression modus.

[11]A programming language is dynamically type safe, if untrapped errors are prevented [27].

[12]In HTML/XHTML only text, i.e. string data, can be passed as hidden parameter, item of a select menu, or value of a radio button or check box. The JSP expression scripting element supports implicit coercion for data that is convertible to String in order to send it to the user agent. However the data is received as string value by the targeted server pages.

[13]The default value of a direct input field must match the input field's type or it must be the empty String.

[14]In HTML/XHTML the tag's value attribute is used for specifying the default value.

2.3.3 NSP Active Single Select Controls

NSP supports single select menus and multiple select menus, which are distinguished as usual by a multiple attribute. A select element has option content elements. The NSP option element has no value attribute, instead of this it has a value content element and a label content element. The value tags contain a Java expression for a selectable data item. The label tags contain a Java String expression that is displayed in the select menu control and denotes the respective data item. In NSP the radio button is supported by an element.

Single select menus and radio buttons are conceptually equal. If a single select control targets a formal parameter that is not an array parameter, it must be ensured that the user actually chooses one of the items. Therefore NSP select menus and radio buttons are active again: a form encompassing such a control cannot be submitted unless the user has finally chosen one of the items[15].

2.3.4 Auxiliary NSP Interaction Controls

NSP supports hidden parameters. Hidden parameters offer a way to overcome the stateless nature of the hypertext transfer protocol [72]. More importantly[16] hidden parameters enable arbitrary reuse of a dialogue method in contexts where only a part of its web signature data should be determined by user interaction.

NSP distinguishes between two check box concepts, that reflect the two different ways in which the HTML/XHTML check boxes are used. First, the check box element yields an actual boolean parameter, true for a checked check box, false for an unchecked check box. Second, a set of value check box elements can be used to offer the user a choice of items. In the value check box tags a selectable data item is given as a Java expression. Assume a set of value check boxes that target the same formal parameter. Such a value check box set is conceptually equal to a multiple select menu, because the user may check a couple of the offered alternatives. A value check box set yields a value array on submit.

[15]Both NSP radio buttons and NSP select menu options have checked attributes. Instead of employing activity for radio buttons and select menus the NSP approach could rely on the request for comment document on HTML [12] that specifies: if no radio button resp. select option is checked, the user agent chooses the first one as pre-selected. Unfortunately a lot of ubiquitous browsers do not implement this behavior. For this reason even the HTML specification [150] differs from [12] in this point. As a result a set of radio buttons or a single select menu may yield no actual parameter as a result. But exactly this must be prevented, if a formal parameter is targeted that requires data entry. There are two other solutions to this problem than active controls. First, an NSP container could dynamically ensure that a checked attribute is added when needed, that is it could simulate the behavior demanded by [12]. Second, the NSP type system could be extended by checks that forces the author to ensure that checked entities always are provided. Anyway single select capabilities with no pre-selected items are desired, simply in all situations where all of the items are equal candidates.

[16]Hidden parameters allow for maintaining state between client/server exchanges. Other opportunities for this are techniques based on client state persistence [105] and URL-rewriting. Anyway today's APIs offer high-level session-tracking mechanisms.

2.3.5 NSP Submit Button

In HTML/XHMTL the submit button can also specify a name/value pair[17]. This is an important facility that is used in web applications to offer the user a choice between different functionalities for the same form[18]. For this purpose the values that represent different functionalities must be defined and the targeted script must switch to the correct alternative on submit. The respective code quickly becomes fault-prone, it's design suffers an "ask what kind"[19] antipattern. In contrast, in the NSP approach it is possible to specify a targeted dialogue method for each submit button. The NSP submit element has an optional callee attribute for this purpose. If no callee-attribute is given for a submit-element, the callee-attribute of the encompassing form is inherited. Furthermore a submit-element can contain hidden-parameters, again NSP gains from its solid theoretical basis: a form can target a variety of different dialogue methods providing different functionality and possessing different web signatures. This allows for flexible and at the same time robust design.

[17]In HTML/XHMTL the submit button is subsumed under the input element.
[18]A form is allowed to have several submit buttons.
[19]We take a generalized view of "don't ask what kind" as explained in 8.1.2.

Chapter 3

NSP Coding Guidelines

This chapter defines the NSP support for gathering array data and data of user defined type. The NSP coding guidelines and rules are elaborated.

The object of the NSP coding guidelines and rules are dynamic document fragments with respect to the generated user interface description. There are two kinds of coding guidelines and rules. The parameter guidelines and rules are dealing with client page type safety, the structure guidelines and rules are dealing with client page description safety. The guidelines are declarative characterizations of valid NSP code, they are informal descriptions of demands placed on NSP code. The NSP coding guidelines are accompanied by NSP coding rules that define how to achieve these demands. The resulting notion of correctness can be checked statically and guarantees client page type and description safety.

The NSP parameter guidelines and rules contribute the adaptation of typed programming discipline to the context of server pages development.

The NSP coding guidelines target the developer: correct NSP code is very natural and the NSP coding guidelines are easy to learn. The NSP coding guidelines and rules are formalized in chapter 7 by a Per Martin-Löf style type system.

In the discussion of coding guidelines passive NSP code is considered as generated code. Such passive NSP code is a variant of XHTML code. It is ensured that generated NSP tags will cause the generation of valid XHTML eventually. The mapping of NSP tags to valid XHTML is given canonically and described further in chapter 5.

3.1 The Object of Parameter Guidelines

In a typed programming language the call to a method must fit exactly the method signature. This notion is picked up but elaborated further due to the special needs of programming a web interface based on a server pages technology. In order to understand the type system of NSP one has to realize that in a scripted server page typically a whole block in the scripting language generates one form in the output. In NSP the static typing rules apply to the whole form, because within the NSP paradigm the whole form is the analogue of only a single method call. As an instructive example for this, have a look at the code fragment in listing 3.1.

Listing 3.1

```
01   {
02     int x;
03     for(int i=0; i<3; i++){
04       x = 810;
05     }
06     m(x);
07   }
08
09 void m(int x){}
```

This is a correct program, though lines 3 and 5 are superfluous and actually can be deleted from an optimizing compiler. In NSP a similar form declaration has to be considered wrong; in the form generated by listing 3.2 the user may enter three int data, however the targeted server page expects exactly one int value. That is, in NSP user interface description code that is created dynamically by a form declaration, cannot be considered just as a block for computing actual parameters comparable to line 3, 4 and 5 in the above code fragment. Instead of this it is considered as an editable method call offered to the user as a whole. It is comparable to line 6 in the above example. Precisely in this sense the form has to support the signature of the called method.

Listing 3.2

```
<form callee="m"><java>
  for(int i=0; i<3; i++){</java>
    <input type="int" param="x"></input><java>
  }</java>
  <submit></submit>
</form>

<nsp name="m">
  <param name="x" type="int"></param>
  <html>
    ...
  </html>
</nsp>
```

The object of the parameter guidelines is dynamic NSP code that generates form content.

Definition 3.1.1 (NSP Parameter Guideline) *An NSP parameter guideline describes, what kind of and how many controls must be provided for what kind of formal parameter. The statement that a certain kind and number of control must be provided for a certain kind of formal parameter means: the body of every form targeting a web signature that encompasses such a formal parameter will always only generate the specified kind and correct number of controls. These controls are said to target the formal parameter in quest.*

In section 3.3 so called parameter rules are given for the parameter guidelines. The NSP parameter rules are the coding rules that help to achieve the parameter guidelines. They describe the effect of conditional control structures, loops, and sequencing of document parts on the guaranteed number of a certain kind of control.

The first fundamental NSP parameter guideline is independent from the type of the targeted formal parameter: a generated control must not target a formal parameter that does not belong to the targeted web signature.

3.2 Basic Parameter Guidelines

In this section the NSP parameter guidelines for formal parameters of basic type are given. NSP distinguishes between basic types, array types and form message types. The notion of form message type is introduced in section 3.5: a user defined type may be explicitly defined as a form message type; then NSP offers tag support for gathering composed data of form message type. In NSP/Java the basic types encompass the Java primitive types and every object type, i.e. every user defined type or arbitrary Java API type. The parameter guidelines explained in this section are summarized in Figure 3.1. Parameter guidelines for arrays are explained in section 3.4.

Formal Parameters of Primitive Type

For a formal int parameter either one type-correct input control, hidden control, single select menu or at least two type-correct radio buttons must be provided.

A form encompassing a type-correct input control cannot be submitted, if the data entered by the user is not a number. Similarly a formal int parameter implicitly requires an entry by the user. The necessary concept of active direct input controls has been explained in section 2.3.1. A hidden control may target a formal int parameter, as long as its contained expression has int as type; such an expression cannot evaluate to a null object, because int is a primitive Java type. For the same reason a single select menu may target a formal int parameter. Alternatively a set of radio buttons may target the parameter. It must contain at least two radio buttons. An int radio button set or single select menu always produces a unique value, because of the concept of NSP active single select controls introduced in section 2.3.3.

A formal int parameter may be constrained to be targeted by an interactive widget. The param tag has a widget attribute, which may be set to required for this purpose. If a widget is required, the targeting control must not be a hidden parameter. Despite this exception the parameter guideline is the same. Requiring an interactive widget is a powerful concept that allows for the precise specification of a dialogue method as editable method call offered to the user by means of a web signature.

The int type is the only NSP/Java instance of the general notion of an NSP direct input supported primitive type. Other language amalgamations may support more primitive types by a direct input widget in a way that follows the parameter guidelines just introduced.

The dual notion of an NSP direct input supported primitive type is that of an NSP ordinary primitive type. The float type is an example for such a type. For a formal float parameter either exactly one type-correct hidden control, single select menu or at least two type-correct radio buttons must be provided.

Formal boolean Parameters

The boolean type needs specific parameter guidelines. It is supported by an instance of the check box mechanism as explained in section 2.3.4. The boolean type is an NSP specific primitive type.

For a formal boolean parameter either exactly one check box or one type-correct input hidden control must be provided. If the widget attribute is set to required, the formal parameter must be targeted by exactly one check box. It is not allowed to use other controls like direct input or a select menu for targeting a boolean formal parameter. We feel that with respect to user interaction the boolean type is inextricably connected to the notion of check box and this is expressed by the current parameter guideline.

Formal Object Type Parameters

Every non-primitive Java type is termed object type. For a formal parameter of object type at most one type-correct hidden control, single select menu or at least two type-correct radio buttons must be provided. Most importantly it is allowed not to provide any control for a formal parameter of object type. If no control is provided, the system will fill in a null object on submission automatically.

For a formal parameter of object type the widget attribute may be set to required, too. Then exactly one type-correct single select menu or at least two type-correct radio buttons must be provided. It is not longer allowed just to omit any kind of control: the purpose of the required widget attribute is to ensure that an interaction capability is offered to the user. In general there is no direct input capability for an object type and possible interaction capabilities are single select menus or a set of radio buttons. However, as opposed to formal parameters of primitive type, requiring a widget does not guarantee valuable data on submission, because nothing prevents a data item provided by e.g. a select menu to be a null object. This leads to another kind of NSP object type.

As for primitive types, it is distinguished between direct input supported object types and ordinary object types. Examples for direct input supported object types in NSP/Java are the Integer type and the JDBC API type Date [186]. For a formal parameter of such type the above parameter guidelines are applicable, except for the possibility to additionally target a parameter by the type-correct input control. Beyond this for such types the formal parameter tag has an entry attribute. This entry attribute may be set to required. Then for a formal parameter exactly one type-correct input control must be provided. Furthermore the system statically ensures that every targeting input control dynamically ensures data entry. As discussed above, only this can prevent a null object from being submitted.

Formal String Parameters

The Java type String is another example for an object type that is supported by an active direct input control. Actually it has three such controls, the usual input field, a password field and the text area control. The parameter guidelines are the same as the ones for types like Integer or Date, up to the additional input controls that are handled the same way as the input field. The only subtle difference concerning the input controls has already been described in section 2.3.1: no null object will be transmitted anytime.

Furthermore it can be specified that a formal String parameter must be targeted by the specific password control or by the specific text area control. The widget attribute is used for this purpose. If it is set to password exactly one password field must be provided, and analogous for the text area widget.

3.3 NSP Parameter Rules

An NSP parameter guideline is a demand on the kind and number of controls generated by a piece of NSP code that targets a certain kind of formal parameter. This section gives rules that define how to achieve these demands.

Determining the number of a certain control in a static part of an NSP document is trivial. But NSP code is a mixture of static and dynamic parts. However, in a valid NSP document only Java code and control tags are relevant with respect to the parameter rules[1], because all the tags concerning layout and the tags that switch between active and passive document parts have no influence on the control number.

Indeed only a Java subset is relevant; the parameter rules can be given with respect to just a few concepts, i.e. sequencing, switch structures and loops.

Consider the NSP code fragment in listing 3.3. Lines 1 to 6 generate four hidden parameters that target the formal parameter p1. Thereby it does not matter that the first two hidden parameters are provided merely by composition of document parts, whereas the latter are interlaced by sequences of Java code.

[1]The object tags explained in section 3.5 are relevant with respect to the parameter rules, too.

Table 3.1: Valid Controls. This table visualizes the NSP parameter guidelines for formal parameters of basic type by specifying kind and number of valid controls for each formal parameter type. For every type only the fields that are filled out are the valid controls. A formal parameter of a certain type must be targeted by one of the valid controls, whereas the specified number must be accomplished. The types int, float and Integer are representatives for the classes of NSP direct input supported primitive types, NSP ordinary primitive types and direct input supported object types, respectively. ObjectType stands for every ordinary object type.

	input field	hidden	single select menu	radio button	value check box	select menu	password	text area	check box
int	1	1	1	≥2					
int widget=required	1		1	≥2					
float etc.		1	1	≥2					
boolean		1							1
boolean widget=required									1
Integer (Date)	0,1	0,1	0,1	0,≥2					
Integer widget=required	1		1	≥2					
Integer data=required	1								
String	0,1	0,1	0,1	0,≥2				0,1	
String widget=required	1		1	≥2				1	
String data=required	1							1	
String widget=password							1		
String widget=textarea								1	
ObjectType		0,1	0,1	0,≥2					
ObjectType widget=required			1	≥2					

Furthermore the switch control structure and the loop must to be taken into account. However selectors, branching conditions, and loop conditions cannot be considered at compile time; the parameter rules must be independent from them. Altogether we need to determine the number of generated controls with respect to the NSP code skeleton in listing 3.4, which consists of relevant structure and elements only. Sure, it is not possible to determine the exact numbers in general, but it is safe, that exactly four hidden parameters are generated that target p1, at most one input control is generated that targets p2[2], at most one input control is generated that targets p3, and an arbitrary number of input controls is generated that target p4.

In Java there exist three different kinds of conditional control structure, i.e. the if-structure, the if-else-structure, and the switch structure. For the purpose of NSP parameter rules it is necessary to distinguish between completed switches and uncompleted switches. A completed switch is one that is ended by a default branch while an uncompleted switch is the opposite. The if-structure can be construed as a uncompleted switch with just one branch. The if-else-structure can be construed as a completed switch with two branches, whereas the else branch becomes the default branch of the switch. As a coding convention, in NSP code it is forbidden to end a switch branch without a break statement[3]. There are three different kinds of loops in Java, the for-loop, the while-loop and the do-while-loop. From the viewpoint of the NSP parameter rules the different kinds of loops are the same. Furthermore the NSP parameter rules do not distinguish between sequencing of NSP blocks that stems from Java code sequencing and such sequencing that stems from composing document parts.

Definition 3.3.1 (NSP parameter rule) *The NSP parameter rules describe how many controls of a certain kind are generated by a piece of NSP code. The number of generated controls is described by a lower and an upper bound on the exact number of generated controls. The NSP parameter rules are given in terms of building blocks and sequences of building blocks. The basic building blocks are the controls in quest. The only further building blocks are completed switch structures, uncompleted switch structures and loop structures. A completed switch is a switch that is ended by a default branch. All case branches and the default branch together are named the branches of the switch. An uncompleted switch is a switch that is not ended by a default branch.*

[2]At most one input control is generated that targets p2. This is an example for a type error: if a formal int parameter is targeted it must be ensured that exactly one appropriate control is provided.

[3]No doubt, it is possible to elaborate rules for switches with a fallthrough facility, but they are unnecessarily complicated. We argue that mixtures of branches that end with a fallthrough and branches that end with a break lay in a gray area between structured and unstructured code. We argue that there are no urgent examples that rely on the use of fallthrough. Interesting examples that rely on fallthrough live in the world of code bumming like e.g. Duff's device [153] (interestingly Duff's device is no valid Java Code [79] anyway). Convenient NSP parameter rules can only be given in terms of completely structured code. For example in every language amalgamation goto statements would be forbidden [54]. Anyway in Java no goto statement is realized (though goto is a reserved keyword).

These are the parameter rules:

- Single control. A single control of a certain kind generates exactly one control of this kind in the output user interface description language.
- Sequence of building blocks. Assume that every building block is safe to generate at least a specific lower bound of a fixed certain kind of control. Then it is safe, that the sequence will generate at least as much controls as the sum of all these single lower bounds. Analogously for upper bounds.
- Completed switch structures. Assume that every branch is safe to generate at least a specific lower bound of a fixed certain kind of control. Then the switch structure is only safe to generate as least as much controls as the smallest of all these single lower bounds. Analogously, if it is safe that every branch generates at most a specific upper bound of a fixed certain kind of control, then it is only safe that the switch structure generates at most as much controls as the highest of all the single upper bounds.
- Uncompleted switch structures. With respect to upper bounds on the number of generated controls the parameter rule for uncompleted switch structures is the same as the parameter rule for completed switch structures. With respect to lower bounds it must differ. Nothing can be assumed about the number of controls of any kind that will be generated at least by an uncompleted switch structure, because in general it is always possible, that none of the branches is selected.
- Loop. If it is safe that a loop body does not generate a certain kind of control, the loop trivially will not generate this control, too. If it is safe that the loop body generates at least one control of a certain kind, it is safe that the loop will generate an arbitrary number of this kind of control. Nothing more can be assumed about the number of controls generated by a loop, i.e. the lower bound is zero and no upper bound can be determined.

From the NSP parameter rules the NSP developer can derive compound rules that define how to achieve the complex demands described by the NSP parameter guidelines. As an example we define how to write code for a form that targets a formal int parameter. The respective parameter guideline prescribes, that such code must always generate either one type-correct input control, hidden control, single select menu or at least two type-correct radio buttons.

If the code is a sequence of blocks, there are three valid possibilities. The first is that exactly one of the blocks generates one type-correct input control, hidden control or single select menu. The second is that at least one of the blocks generates at least two radio buttons. The third is that at least two blocks generate at least one radio button. If the code is a completed switch, all branches of it must provide the correct number of valid controls. Controls targeting the formal parameter in quest must not occur in a branch of an uncompleted switch that is contained in the code, because it is not decidable whether such a switch generates the control or not. Similarly such a control must not occur inside a loop that is contained in the code. The latter might seem to be the most severe constraint, but we argue that it does not prevent the developer from writing code for all reasonable applications. Placing a control inside a loop obviously has the reason of gathering array data; NSP parameter guidelines for arrays are given in section 3.4.

Listing 3.3

```
<hidden param="p1">new Person();</hidden>
<hidden param="p1">new Person();</hidden><java>
x:=1;</java>
<hidden param="p1">new Person();</hidden><java>
x:=2;</java>
<hidden param="p1">new Person();</hidden><java>
switch (y) {
  case (y==1) : </java>
    <input type="int" param="p2">
    </input><java>;break;
  case (y==2) : </java>
    <input type="Integer" param="p3">
    </input><java>;break;
  case (y==3) : for (int i=0; i<fooBound; i++) {</java>
                  <input type="String" param="p4">
                  </input><java>
                };break
}</java>
```

Listing 3.4

```
<hidden param="p1">new Person();</hidden>
<hidden param="p1">new Person();</hidden>
<hidden param="p1">new Person();</hidden>
<hidden param="p1">new Person();</hidden>
switch () {
  case () : <input type="int" param="p2">
  case () : <input type="Integer" param="p3">
  case () :
     for () {
       <input type="String" param="p4">
      }
}
```

3.4 Parameter Guidelines for Arrays

A web signature may encompass formal array parameters. A formal array parameter may be targeted by an arbitrary mix of controls, as long as all controls are type-correct and valid with respect to the contained items' type. The valid controls for a formal array parameter encompass the controls that are valid for a completely unconstrained formal parameter[4] of the respective contained items' type. Furthermore type-correct value check boxes and select menus are valid controls for array parameters. The number of controls is not constrained for all kinds of controls, except for radio buttons. If a radio button targets a formal array parameter there must be at least a second radio button targeting the same parameter. The valid controls for the several array types are summarized in Figure 3.2.

For example, a formal int array parameter may be targeted by an arbitrary mix and number of type-correct input controls, hidden controls, single select menus value check boxes, select menus and perhaps a set of at least two radio buttons.

A formal parameter of array type is optional. If no control targets the parameter the system will automatically pass a null to the dialogue method on submit. It is not possible to give requirements for a formal array parameter. The emphasis of the NSP array mechanism is on the support for gathering array data in forms. If a constrained dynamic data structure is desired, complex message types and their respective facilities described in section 3.5 may be used.

NSP offers sophisticated support for gathering array data in forms. Listing 3.5 shows a form that targets a dialogue method that excepts an Integer array x and an int array y. Nine input fields are generated for the parameter x. Input widgets for Integer values are optional by default. If the user does not enter a value into a field this field is just ignored. All fields that have been received valid user data together provide the submitted actual array parameter. If the user did not enter any data a null object will be provided by the system for the parameter x on submission. Another nine input fields are generated for the parameter y. Input widgets for int values are required by default. The user must enter valid data in all the fields, otherwise the form cannot be submitted. The submitted array is guaranteed to have a length of nine.

As a second opportunity it is possible to explicitly specify the index of an array item. For this purpose the input field tag may contain an index element. The content of such an index element must be a Java int expression. If explicit definition of array element indexes is chosen, the specification should be unique. Furthermore the specification should be complete. Array items, that have a redundant index or don't have an explicit index are attached arbitrarily to the array.

Listing 3.6 is oriented towards the example from listing 3.5. A loop generates nine input fields for the parameter x. This time fields not filled out by the user,

[4]An unconstrained formal parameter is a parameter for which no widget is required or specified and no entry is explicitly required.

Listing 3.5

```
<form callee="N"><java>
  for {int i=0; i<9; i++){</java>
    <input type="int" param="x"></input><java>
  }
  for {int i=0; i<9; i++){</java>
    <input type="Integer" param="y"></input><java>
  }</java>
  <submit></submit>
</form>
```

cannot simply be ignored. Instead of this a null objects are inserted at the specified positions. Another input field is generated behind the loop. It has index 10. There is no input field with index 9 targeting the parameter x. If the form is submitted, again a null object will be inserted as ninth element. Another ten input fields are generated for the parameter y. These fields must be filled out. Again there is no input field with index 9 targeting the parameter y. If the form is submitted, this will result in an dynamic type error, because the system must fail to insert an appropriate value for this missing element of primitive type.

Listing 3.6

```
<form callee="N"><java>
  for {int i=0; i<9; i++){</java>
    <input type="Integer" param="x">
      <index> i </index>
    </input><java>
  }</java>
  <input type="Integer" param="x">
    <index> 10 </index>
  </input><java>
  for {int i=0; i<9; i++){</java>
    <input type="int" param="y">
      <index> i </index>
    </input><java>
  }</java>
  <input type="int" param="x">
    <index> 10 </index>
  </input>
  <submit></submit>
</form>
```

Table 3.2: Valid Controls. This table visualizes the NSP parameter guidelines for formal array parameters.

	input	hidden	single select menu	radio button	value check box	select menu	password	text area	check box
int [ ]	×	×	×	×	×	×			
float [ ] etc.		×	×	×	×	×			
boolean [ ]		×							×
Integer [ ] (Date)	×	×	×	×	×	×			
String [ ]	×	×	×	×	×	×		×	
ObjectType [ ]		×	×	×	×	×			

3.5 Parameter Guidelines for Form Messages

In NSP objects of user defined types can be used as hidden parameters, select menu items, and radio button or check box values. But NSP offers tag support for gathering data of user defined type in forms, the so called NSP form message mechanism. Input or selection capabilities for the single attributes of an object or of a complex object net may be offered to the user. Thereby a special object element, that resembles the with construct in MODULA-2 [187], enables the construction of data records[5].

In order to be supported in the way described above a user defined type must be explicitly marked as a form message type[6]. In general a form message is a record of attributes. The attributes are the fields that are supported by the form message mechanism. The user defined type underlying a form message type may have auxiliary fields[7]. Therefore, for every programming language amalgamation a naming convention must be defined for form message types, that distinguishes attributes from auxiliary fields. In NSP/Java the naming convention of the Java Beans [85] component model is chosen. A form message type is a Java Bean. A form message type attribute is a Java Bean property.

In this chapter we visualize form message types as UML class diagrams [13]. These diagrams are drawn within a defined perspective, which is a kind of specification perspective in the sense of [74][41]. The perspective abstracts away

[5]Similarly the group element of the XForms[69] technology enables the construction of semi-structured data entered by the user.

[6]In a concrete language amalgamation marking a user defined type as a form message type may be defined by a naming convention or as implementing a marker interface.

[7]So far visibility mechanisms are not an issue, they must be considered with respect to concrete programming language amalgamations only.

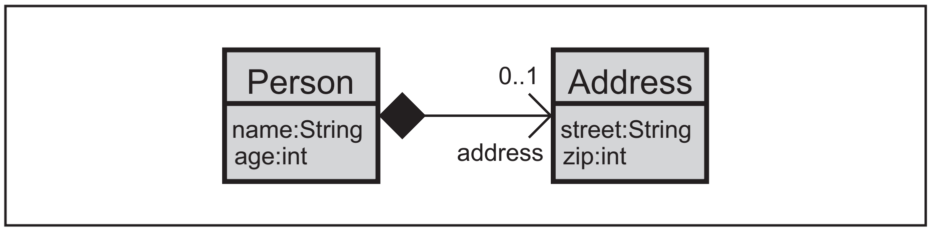


Figure 3.1: Example Form Message Type.

from possible concrete naming conventions. Only form message type attributes occur in the diagram, i.e. auxiliary fields are not visualized. Attributes that have form message type are visualized as associations, all other form message type attributes are visualized as UML object attributes. The associations are navigated and carry the attribute's name as role name. No methods occur in the diagrams. For instance, with respect to the NSP/Java naming convention, a class denotes a Java Bean, object attributes and associations denote Java Bean properties. In the diagrams all associations are compositions in order to emphasize, that all object nets gathered by the user are trees[8].

A first introductory example is given by the form message type Person in Figure 3.1 and the form in listing 3.7. The example is already sufficient in order to state the NSP parameter guidelines for form message types. The form targets a web signature that consists of a formal parameter customer of type Person. Within the form data for an object net consisting of a Person object and an Address object may be entered; on submit the respective object net is constructed[9]. A formal parameter of form message type must be targeted by at most one object element. Like the control tags, the opening object tag has a param attribute for this purpose. The object element contains controls and possibly further object elements for the attributes of the targeted parameter. Thereby the NSP parameter guidelines apply recursively, i.e. from a form's viewpoint a form message type can be understood as a nested formal parameter type.

The form in listing 3.8 targets a web signature that consists of a formal array parameter customers, the array items' type is again the type Person given in Figure 3.1. A formal array parameter of form message type may be targeted by an arbitrary number of object elements[10], as long as the NSP parameter guidelines are fulfilled recursively for each object element. The single object nets may be explicitly indexed. An object element may contain an index element for

[8]Following [156] aggregations may form cycles, but constrain the respective link relationship to be transitive and antisymmetric. Compositions are aggregations with an additional constraint: a part may only be part of one composite.

[9]More concrete, on submit a Person object and an Address object are created, thereby a pointer to the Address object is created which automatically becomes the address attribute of the Person object.

[10]Figure 3.8, line 3, line 9

Listing 3.7

```
<form callee="target">
  <object param="customer">
    <input type="String" param="name"></input>
    <input type="int"    param="age"></input>
    <object param="adress">
      <input type="String" param="street"></input>
      <input type="int"    param="zip"></input>
    </object>
  </object>
</form>

// web signature of target
<param name="customer" type="Person"></param>
```

this purpose[11].

An object element may be given a uniquely identifying name with the optional id attribute[12]. Then controls and object elements for gathering data for form message type attributes can occur anywhere. They have to reference the object element they belong to by its identifying name. Opening control[13] and object tags[14] have an optional in-attribute for this purpose. Identifying and referencing object elements provides a convenient way for allowing arbitrary form layout. With object element nesting only, some desired occurrences of controls that contradict the rigid structure of layout elements could not be realized[15]. Listing 3.8 yields an example. Data for two object nets are gathered. Data for the attributes of a each object net are gathered in one table column at each case. Thereby the referencing mechanism just introduced is needed[16].

A formal parameter of form message type is optional. It is possible not to provide an object element for it. On submit the system will pass a null object to the receiving dialogue method. Again it becomes important, that the notion of form message type attribute is subsumed under the notion of formal parameter. For example the first object element in line 3 of listing 3.8 possesses an object element for the Address attribute in line 7, whereas the second object element in line 9 does not posses an object element for this attribute. Most importantly, though a formal parameter of form message type is optional, this is not carried over to its contained attributes. An object element may not be provided, but if it is provided, the NSP parameter guidelines apply to the contained attributes,

[11]Figure 3.8, line 4, line 10

[12]Listing 3.8, line 3, line 9

[13]Listing 3.8, e.g. line 18

[14]Listing 3.8, line 7

[15]The XForms technology even introduces decoupling of controls from forms [69] for similar reasons. The respective attributes are the id-attribute and the ref-attribute.

[16]A solution based on nested tables cannot achieve the same layout effect. If the data cells of the given minimal example contain more information, so that data cells of a row have different heights, correct alignment is not ensured any more.

Listing 3.8

```
<form callee="target">
 <table>
  <tr><td><object param="customers" id="first">
       <index>1</index>
       <input type="String" param="name"></input>
      </object>
      <object param="adress" id="firstPart" in="first"></object>
    </td>
    <td><object param="customers" id="second">
       <index>2</index>
       <input type="String" param="name"></input>
      </object>
    </td>
  </tr>
  <tr><td><input type="int" param="age" in="first"></input></td>
    <td><input type="int" param="age" in="second"></input></td>
  </tr>
  <tr><td><input type="String" param="street" in="firstPart"></input>
    </td>
    <td></td>
  </tr>
  <tr><td><input type="int" param="zip" in="firstPart"></input>
    </td>
    <td></td>
  </tr>
 </table>
</form>

// web signature of target
<param name="customers" type="Person[]"></param>
```

for example an int attribute would be required.

For formal parameters data entry may be specified as required with the data-attribute[17]. For a parameter of form message type this enforces that exactly one object element for it is provided.

The fact that formal parameters of form message type are optional enables the support of cyclic user defined data in forms. The form in listing 3.9 targets a web signature that consists of a formal parameter of type ParticipantList. The type ParticipantList, which is a dynamic data structure, is given in Figure 3.2. In the example form input capabilities for three list elements are given. On submit an object net of three list elements is created. For the next-pointer of the third element a null object is filled in. Dynamic creation of input capabilities for dynamic data structures in a statically type-safe manner is possible by recursive

[17]Listing 3.10, line 2

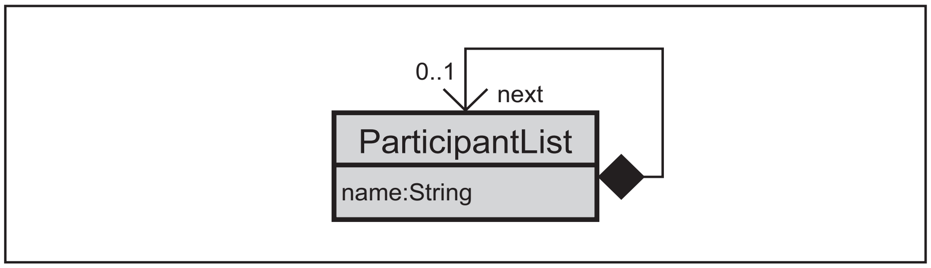


Figure 3.2: Example Cyclic Form Message Type.

definitions of dialogue submethods, which are introduced in section 4.2.

Listing 3.9

```
<form callee="target">
  <object param="participant">
    <input type="String" param="name"></input>
    <object param="next">
      <input type="String" param="name"></input>
      <object param="next">
        <input type="String" param="name"></input>
        <object param="next">
          <input type="String" param="name"></input>
        </object>
      </object>
    </object>
  </object>
</form>

// web signature of target
<param name="participants" type="ParticipantList"></param>
```

NSP defines a fine granular mechanism for putting constraints on the attributes of user defined data. For this purpose a parameter-element that defines a formal parameter of form message type may contain constraints-elements. Each of the constraints-elements must uniquely refer to a form message type that is involved as a part in the definition of the formal parameter type. A constraints-element contains a param-element for every attribute of the form message type it refers to. The param-element can be used to pose on the attributes one or several of the constraints that have been introduced in section 3.2[18][19].

Listing 3.10 gives examples for constraints on user defined data attributes.

[18]data=required, widget=required, widget=password, widget=textarea

[19]The mechanism is not recursive. The param-elements of a constraint specification must not contain further constraints-elements.

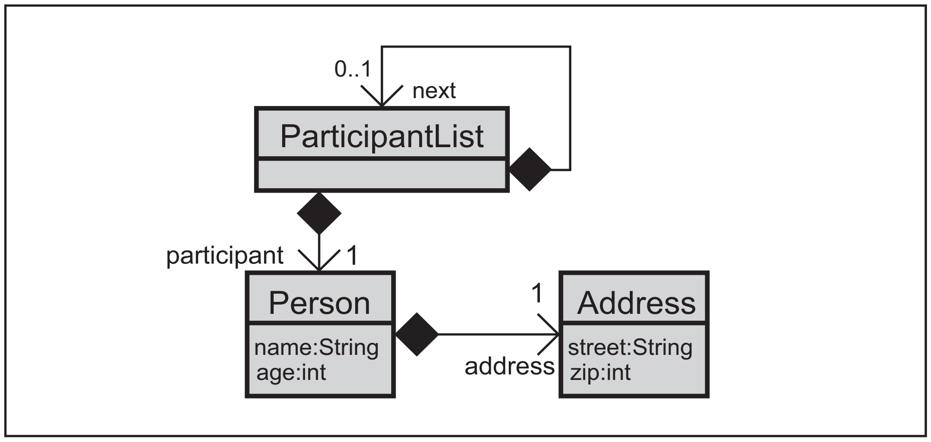


Figure 3.3: Example Complex Form Message Type.

Consider the form message type given in Figure 3.3. Again it is ParticipantList - like in Figure 3.2 - but this time the type's attribute has a form message type. The attribute's type is Person, which already served as an example before (Figure 3.1), but this time the cardinality of the address-association has changed from 0..1 to 1, which depicts that the address attribute is required this time. In listing 3.10 a formal parameter of type ParticipantList is defined. The constraint specification in line 3 to line 6 causes, that the participant-attribute is required. That is, whenever an object element for gathering data for a list element for the actual parameter is generated, it must contain an object element for its participant-attribute. There is no additional constraint posed on the next-attribute. The next-attribute must be optional, because it forms a cycle in the class diagram. Further constraint specifications ensure for example, that for every Person object, the user has to enter a name, a street and a zip code. Note that it would not be sufficient to constrain the street-attribute and zip-attribute of the type Address to be required in order to achieve this[20]. The address-attribute of the type Person must be required for this purpose, too[21]. Without requiring the address-attribute, the whole address is optional. Requiring street and zip code only, just enforces that these are required in the case that an object element for the address is actually generated.

3.6 Document Structure Guidelines and Rules

The NSP coding guidelines for ensuring client page type safety has been described in sections 3.2 to 3.5, this section completes the discussion by describing remaining coding guidelines and rules concerning client page description safety.

[20]Listing 3.10, line 13, line 14
[21]Listing 3.10, line 10

Listing 3.10

```
// web signature of target
<param name="participants" type="ParticipantList" data="required">
 <constraints type="ParticipantList">
   <param name="participant" data="required"></param>
   <param name="next"></param>
 </constraints>
 <constraints type="Person">
   <param name="name" data="required"></param>
   <param name="age"></param>
   <param name="address" data="required"></param>
 </constraints>
 <constraints type="Address">
   <param name="street" data="required"></param>
   <param name="zip" data="required"></param>
 </constraints>
</param>
```

The first basic structure guideline demands that an NSP server page always only generates well-formed XML. The corresponding coding rule for NSP code states the following: the NSP document must be a well-formed XML document and the Java block structure must be compatible with the XML block structure. The latter means that for every XML tag the corresponding dual tag must occur in the same block, whereas as a necessary exception to this the Java tags are ignored. Java tags must be considered merely as switches between the programming language and the markup language with respect to the document structure[22] [23].

The second structure guideline demands that an NSP server page always only generates valid passive NSP/Java code. That means that an active NSP document fragment must only generate valid content elements with respect to its directly encompassing tags. Passive NSP code is essentially XHTML up to the following differences:

- some element properties are supported by content elements instead of attributes,
- some elements, e.g. the form element and the control elements, are modified,
- some new elements are introduced, e.g. the param element and the call element,

[22]In technical terms java opening and closing tags are go-betweens of different lexical states.

[23]Alternatively the given coding rule can be reformulated more concisely in the following way: The document must be well-formed XML. Then Java tags are ignored, instead every Java block is considered a new document element. The document in quest must be well-formed with respect to the resulting element set.

- some content elements are not required, though their counterparts are required in XHTML.

The first three differences pose no problems. The introduced modifications are resolved when the final page, i.e. the XHTML page that is sent to the browser, is generated. The fourth difference is discussed in the sequel, thereby the coding guideline must be discussed with respect to the finally called page.

In XHMTL [172] some elements require that a certain kind of content element appears at least once. In NSP these demands are dropped for pragmatic reasons, not for technical reasons. For example in XHTML a list must contain at least one item. A table must contain at least one row. A table row must contain at least one table data cell. A select list must contain at least either one option or one group of option. However most of these constraints are artificial[24], because browsers can, and current browsers do, cope with violations of these constraints in a natural sensible way. For example, lists without items, tables without rows and table rows without data cells are just not displayed. In NSP it would be possible to define coding rules that ensure validity with respect to the aforementioned constraints, but this would lead to unjustifiably complex demands on the NSP code. For example, list items are typically provided by a loop and in such a case the developer would have to add for example an initial list item by sequencing. Therefore the NSP concept of client page description safety is weaker than full XHTML validity[25].

The coding rule for the second structure guideline is given, though a bit verbose, with respect to sequencing, switch structures, and loops again. The demands are irrespective of the parameter guidelines and rules given so far. Pure Java code is considered neutral. In a sequence of document parts, all document parts must be neutral or generate valid elements. In a loop the body must be neutral or generate valid elements. In a switch structure all branches must be neutral or generate valid elements.

[24]Moreover some of these constraints are ineffectual. For example though an item is required for a list, it is not required that a list must not be empty. As another example, an interesting constraint for table rows is not that a row contains at least one data cell, but that all rows of a table contain the same number of data cells up to grouping with the column-span mechanism.

[25]Note that NSP client page description safety is partly stronger than XHTML validity as well, e.g. at least two radio buttons are needed in order to form a valid composed radio button control in the NSP approach.

Chapter 4

Web Presentation Layer Architecture

In this chapter we provide a discussion of important current approaches to web interface programming based on the Model 2 architecture [59]. From the results we derive how to improve web presentation layer architecture. Enabling technology for this is the NSP concept of typed server side calls to server pages. The concept of higher order server pages is introduced, which enables even more flexible design.

The central architectural questions concerning web based system interfaces are located on the server side. We review current web application frameworks for building dynamic web pages. Web application frameworks consider only the presentation layer in a multi-tiered web application. Our considerations are based on an analysis of the problem addressed by these frameworks. Special attention is paid to proposed composition mechanisms. In that comparison we can analyze the technological contributions as well as the shortcomings of these approaches.

4.1 Model 2 Architecture

In practice the tight coupling of code with layout has become a drawback for server pages technology. Therefore, separation of business logic processing and presentation generation, called processing/presentation separation in the following for short, became a goal.

In the discussion on how to reach processing/presentation separation, Sun has become influential by proposing several server side architectures, therein the "redirecting request" application model - coined Model 2 architecture afterwards [143]. This model has become commonly known as following the Model View Controller paradigm. We will in due course outline that it is a misconception about Model View Controller if the Model 2 architecture is subsumed under this pattern. We therefore give an evaluation of the Model 2 approach

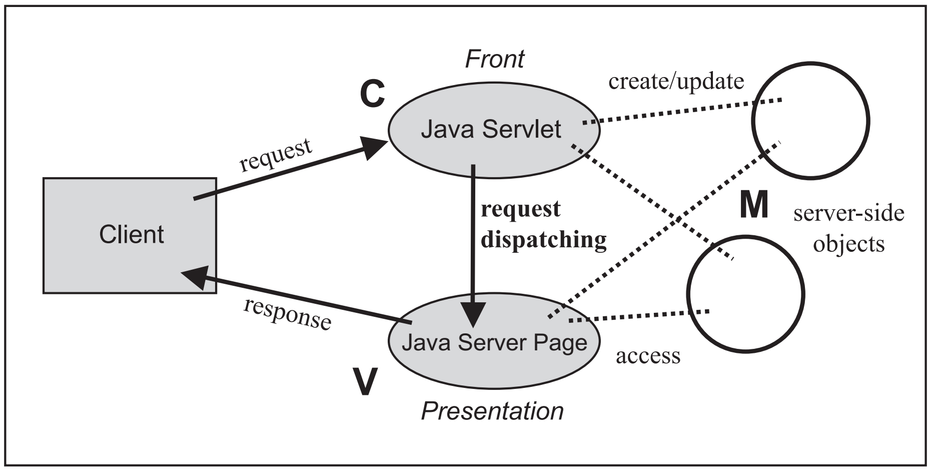


Figure 4.1: Model 2 Architecture. The figure visualizes the "redirecting request" application model coined Model 2 architecture. The model has become commonly known as following the Model View Controller paradigm. The server side objects are considered as model (M), the front components as controllers (C), and the presentation components as views (V).

without relying on the MVC argument.

The Model 2 architecture uses a threefold design in which the request is first directed to a front component, typically a servlet, which triggers the creation of a content object, typically a Java bean (Figure 4.1). The bean is then passed to a presentation component, typically a scripted server page where the data within the bean is embedded in HTML/XHTML. For Model 2 architecture some good practices are established on how to partition the request processing between the three parts. The most important recommendation is related to the use of the server pages: the server pages shall be used only for presentation purposes. Model 2 architectures can achieve a reuse of presentation components. If several front components generate under certain conditions the same output page, this page can be used from both components. Model 2 also allows separate maintenance of totally different response pages that may be generated from the same front component under certain conditions.

The Struts [50] framework is widely accepted as the open source reference implementation of the model 2 architecture. Struts proposes functional decomposition based on a proprietary composition approach in which business processing units do inform the controller object about the next processing step. Parameter passing between processing units is not established by the Java method parameter passing mechanism, but by emulating a parameter passing mechanism through transferring bean objects.

It is important to clarify a serious misunderstanding in architecture proposals for web site development. The web application frameworks following the Model

2 approach do not follow the Model View Controller paradigm. Model View Controller (MVC) [104] was introduced in Smalltalk and is a completely different concept. It only has superficial similarities in that it has three components from which one is related to the user interface, another to the application. However, the problem solved by the MVC paradigm is totally different. MVC is related to event notification problems within a GUI that provides different views on the same data, which have to be synchronized. MVC is renamed within the pattern community as observer pattern [75] and became an accepted general pattern for event model design problems[1]. The misnomer is even more astounding if one considers that the property of GUI's which makes MVC necessary, namely view update, i.e. push technology, is well known to be absent in the pull based approach of HTML/XHTML browsers.

The fact that web application frameworks rely on a misconception of the MVC paradigm does not necessarily imply that these frameworks have a bad design. But the argument for this architecture, namely that it follows a proven good design, is flawed. Only by recognizing that this argument is invalid the way is free for a new evaluation of the architecture and a recognition of advantages as well as drawbacks.

The Model 2 architecture defines a fixed decomposition combined with an intended separation of concerns. The incoming request is performed on the business model, then data are presented to the user in response. The difficulty with the approach lies not in the proposals for separation of concerns, but with the composition mechanism offered. The question is which semantics governs the interplay between the components: after you know how to divide, you have to know how to conquer.

The Model 2 architecture offers a complex communication mechanism based on the passing of beans. Beans are attached to a hashtable by the generating unit and retrieved from the hashtable by the target unit. In that way data is transmitted from the servlet to the scripted page. This mechanism is nothing more than a parameter passing mechanism, but without static type safety. The semantics of the composition paradigms of presentation and business logic is only conceivable by direct reference to the components found in the running system. In contrast, we will later use our NSP approach, where simple method call semantics is sufficient and allows for a sound architecture. Hence in the Model 2 architecture a considerable part of the architecture redefines a parameter passing mechanism which delivers no added value beyond method invocation. The Model 2 architecture therefore is still interwoven with a legacy technology driven design pattern that is far from creating a clear cut abstraction layer.

[1]The Java Beans component model relies on an observer pattern based event model. The GUI event model of the first Java version 1.0 followed the "chain of responsibility" design pattern [80], today it follows the observer pattern.

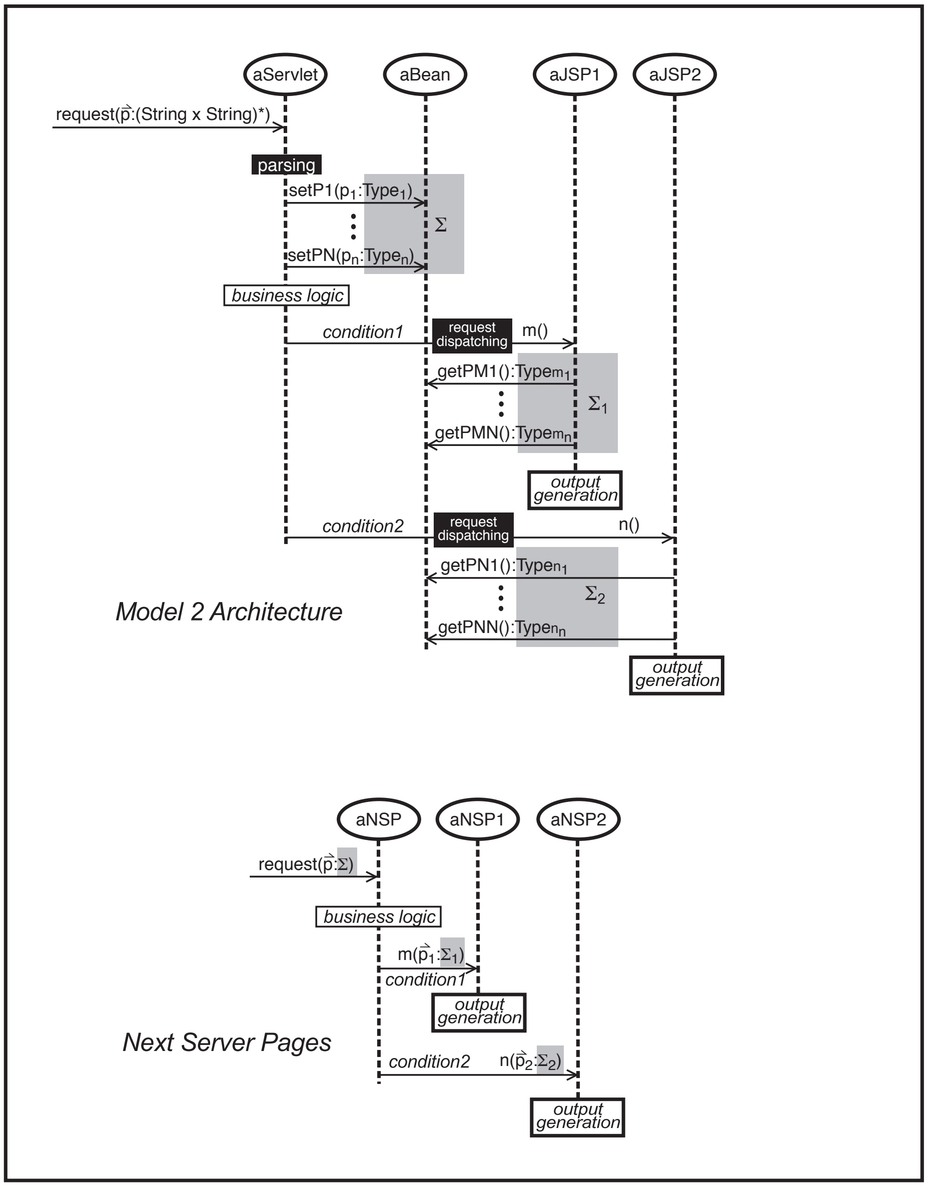


Figure 4.2: Model 2 Architecture versus NSP Functional Decomposition. The figure shows a typical control and data flow in a Model 2 architecture system up to details of request dispatching and the improvement of a counterpart system build on Next Server Pages technology by an interaction diagram.

4.2 NSP Functional Decomposition

NSP is open with respect to architectural decisions. NSP distinguishes between server pages that may be called across the net by a form or a link and server pages that may be called by another NSP server page on the server side in order to be included. The latter server pages has been termed dialogue submethods before. The NSP call mechanism for dialogue submethods has identical semantics as the Java method call with respect to parameter passing. It is the only composition feature that is needed to build sound and well understood web application architectures. NSP does not force the user into a specific design. With NSP no early decision between Model 1, Model 2 or other architectures is necessary.

The NSP approach to design can be seen as generalization of another proposal of the JSP specification, named "including requests" [143].

In order to call a server page from within another server page, the call-element is used[2]. The opening call-tag has a callee-attribute. Upon call the targeted server page generates a document fragment that replaces the respective call-element in the calling document. The output of the dialogue submethod must be a valid content element with respect to the context of the respective call element. Actual parameters are given to the called dialogue submethod by actparam-elements. The opening actparam-tag has a param-attribute, which has the same purpose as the the param-attributes of the NSP control elements. For each formal parameter of a dialogue submethod exactly one actual parameter must be given in every targeting call-element. Call elements may only contain actparam-elements, especially they cannot contain dynamic code[3].

The example in listing 4.1 consists of a main page and two dialogue submethods. The first dialogue submethod receives a String parameter and an int parameter and produces a message with respect to these parameters. The second dialogue submethod is called inside a table element and receives an array of Article objects. An Article object has three String properties x,y, and z. The dialogue submethod generates a table row for each object. A row contains three table data cells, one for each object property.

In the JSP technology parameter passing to a JSP differs fundamentally whether the JSP is called across the net or called on the server side. In the first case, parameters come as raw string data, as it is inherited from the old CGI mechanism. However, if a server page is called locally, it is established coding practice to pass the parameters by a bean object attached to the request parameter. Hence, a page must be designed either to be callable from the net or to be callable from the server and in both cases the developer has to face a parameter passing mechanism different from any reasonable parameter passing

[2]Listing 4.1, line 8-11, line 14-16

[3]The call element provides the NSP equivalent to the JSP request dispatching include mechanism. Analogously NSP supports redirect and server-side redirect, i.e. forward, by appropriate elements in the same way as the call element.

Listing 4.1

```
<nsp name="mainPage">
  <html><head><title>Some Page</title></head>
    <body><java>
      String customer;
      int age;
      Article[] articles;
      // get data for variables customer, age, and articles </java>
      <call callee="prelude">
        <actparam param="customer">customer</actparam>
        <actparam param="age">age</actparam>
      </call>
      <table>
        <tr> <td>X</td> <td>Y</td> <td>Z</td> </tr>
        <call callee="tableContent">
          <actparam param="articles">articles</actparam>
        </call>
      </table>
    </body>
  </html>
</nsp>

<nsp name="prelude">
  <param type="String" name="customer"></param>
  <param type="int" name="age"></param>
  <include>
    Hello Mr. <javaexpr>customer</javaexpr> !
    <!-- other ouptut with respect to customer and age -->
  </include>
</nsp>

<nsp name="tableContent">
  <param type="Article[]" name="articles"></param>
  <include><java>
     for (i=1;i<articles.lentgth;i++) {</java>
       <tr>
         <td><javaexpr>articles[i].getX()</javaexpr></td>
         <td><javaexpr>articles[i].getY()</javaexpr></td>
         <td><javaexpr>articles[i].getZ()</javaexpr></td>
       </tr><java>
     }</java>
  </include>
</nsp>
```

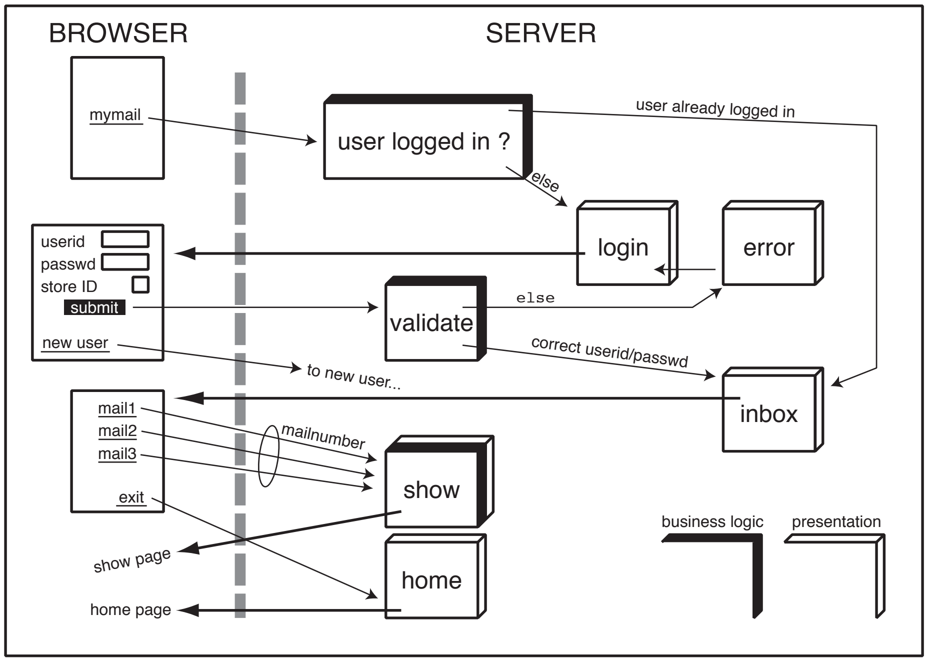


Figure 4.3: Example Interaction Diagram. The figure shows the login dialogue of a web based mail account. The user logs in and views her inbox. If she stores her password, for a certain time no login is necessary.

mechanism[4]. In NSP in contrast parameter passing is identical whether the page is called over the net or within the server. In both cases the parameter passing mechanism is essentially identical to the parameter passing encountered in Java. The parameters of a page in NSP behave identical to local variables in the Java code, in fact they are local variables initialized by the actual parameters. The difference is visualized in Figure 4.2. It follows from the explanation of the NSP implementation in chapter 5 that this transparency in the parameter passing mechanism comes at virtually no additional cost compared to the approaches in web application frameworks.

NSP allows for arbitrary application architectures based on functional decomposition. NSP frees the developer from considering the implementation details of the parameter passing mechanisms. Hence all special runtime entities that are needed in NSP to deliver the method call semantics are hidden from the developer. Processing/presentation separation is in first place a pattern for source code organization. NSP allows to solve the challenges in process-

[4]In the Servlet request dispatching mechanism, it is possible to attach new name/value pairs to the request object URL before invoking another Servlet. The JSP technology provides a JSP standard action, i.e. the param-action, for attaching new arguments for included server pages. However both of these are no parameter passing mechanisms, because only string parameters can be attached in an uncontrolled manner.

ing/presentation separation without referring to system architecture. In contrast, in NSP the functional decomposition mechanism allows for the desired separation of concerns. In Figure 4.3 we give an interaction diagram which shows the login dialogue of a web based mail tool. The user logs in and views her inbox. If she stores her password, for a certain time no login is necessary. In the given example the depth of decomposition is adapted according to the complexity of the respective functionality. The login screen is used for the initial login screen as well as for the login screen after an invalid login attempt. Viewing a mail is realized as a simple server page call. The example demonstrates the openness of NSP for different architectures.

4.3 Higher Order Server Pages

NSP formal server page parameters may receive server pages again. This introduces the notion of higher order server pages. The higher order server pages concept can be exploited to foster system maintainability and system part reusability.

NSP introduces the Page type as single proprietary type to be used for a formal parameter in addition to the types of the amalgamated programming language. Formal parameters of Page type may be used as callee-attributes of opening forms, links, and calls. For a formal Page parameter of a dialogue method either one hidden control, single select menu or at least two radio buttons must be provided in every form targeting the method. The value provided by the control must be the name of an existing server page that belongs to the system or again a formal Page parameter. A web signature with a formal Page parameter may be equally targeted by a hyperlink or, if the web signature belongs to an include server page, by a server-side call, with appropriate usage of hidden parameters respective actparam-elements.

Listing 4.2 gives a simplified example of a typical dialogue cycle:

input page - server-side validation - error page

A registration page includes a registration form by a server-side call. The registration form offers a text input field. The default value of this field is a parameter of the registration form submethod. The registration page chooses an empty string as default for the input field. Importantly the form provides its encompassing include server page as actual parameter. It is used by the targeted server page as error page. On submit the targeted server page first checks a business rule concerning the customer name. If no error occurred, the customer name is processed and the dialogue is continued by forwarding to another server page. Otherwise the form that targeted the server page is redisplayed and serves as a simple error page. This time the last user's input for the customer name is the default value for the input field. The example given in listing 4.2 is an instance of a more general, common design problem that can be given a reusable, flexible solution based on a higher order server page concept. Consider the case that several forms target the same dialogue method, which

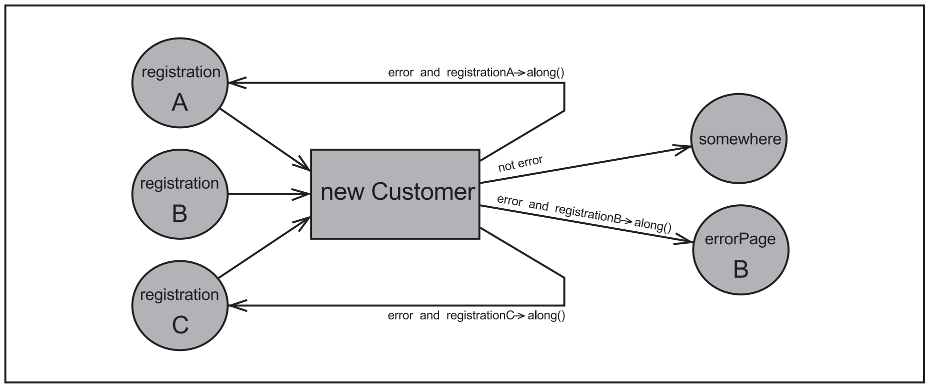

registration
A
registration
B
registration
C
new Customer
somewhere
errorPage
B
error and registrationA->along()
not error
error and registrationB->along()
error and registrationC->along()

Figure 4.4: Example Form Chart Diagram. The user dialogue given by a form chart in this figure poses a typical design problem that can be given a reusable, flexible solution based on a higher order server page concept.

processes the data and branches the dialogue flow, whereas the next page in the dialogue depends on the form that triggered the server page. Without higher order server pages the developer must explicitly keep track of the dialogue and must switch to the correct next page accordingly, which is an instance of a design that suffers an "ask what kind" antipattern[5]. Figure 4.4 shows a feature [58] containing three registration pages similar to the one given in listing 4.2. The feature is visualized as a form chart [60][61]. The server action for processing new customer data validates submitted user entry and presents an error page to the user if necessary. In the case that the action has been triggered by the first or third registration page, the respective page is redisplayed[6]. In the case that the action has been triggered by the second registration page, a specific error page is presented to the user.

[5]Our view of "don't ask what kind" is discussed in 8.1.2.

[6]Dialogue Constraint Language [60] is used to express the so called flow conditions in Figure 4.4.

Listing 4.2

```
<nsp name="Registration">
  <html>
    <head><title>Registration</title></head>
    <body>
      <call callee="RegistrationForm">
        <actparam name="defaultCustomer"> "" </actparam>
      </call>
    </body>
  </html>
</nsp>

<nsp name="RegistrationForm">
  <param name="defaultCustomer" type="String"></param>
  <include>
    <form callee="NewCustomer">
      <input type="String" name="customer">defaultCustomer</input>
      <hidden name="errorPage">RegistrationForm</hidden>
    </form>
  </include>
</nsp>

<nsp name="NewCustomer">
  <param name="customer" type="String"></param>
  <param name="errorPage" type="ServerPage"></param>
  <java>import myBusinessModel.CustomerBase;</java>
  <html>
    <head><title>NewCustomer</title></head>
    <body><java>
      if (CustomerBase.validate(customer)) {
        CustomerBase.createCustomer(customer);</java>
        <forward callee="Somewhere"></forward><java>
      } else {</java>
        <call callee="errorPage">
        <actparam name="defaultCustomer"> customer </actparam>
        </call><java>
      }</java>
    </body>
  </html>
</nsp>
```

Chapter 5

Operational Semantics of NSP

In this chapter an operational semantics of NSP is given by describing a transformation of NSP components to JSP [143] pages, JavaScript [73][130] and Java 2 Platform API [166] objects. The operational semantics considers only NSP components that form a correct system with respect to the static semantics of NSP.

Basic Transformations. An NSP server page is translated into a JSP server page of same name up to technical details like renaming the file's ending. The resulting document is pure XHTML up to the exceptions described in section 3.6 plus Java code inside scriptlet signs. For web server pages the result is a complete XHTML page, for include server pages it is a valid XHMTL fragment, that may occur somewhere in a document body. In NSP both programming language declarations and programming language code are placed inside the same kind of tag, they are distinguished by their occurrence. NSP declarations are mapped to JSP declaration scripting elements, NSP Java code is mapped to JSP scriptlets. NSP Java expressions are mapped to JSP expression scripting elements.

The transformation resolves the differences between NSP and XHTML. An element that represents an XHTML attribute is mapped to this attribute in the respective encompassing element.

An NSP form is mapped to a JSP form. In case that the NSP form does not contain a submit button that specifies a callee different from the encompassing form's callee, the callee-attribute is mapped to the XHTML form's action-attribute. Thereby the attribute's value must be added to a given constant URL representing the site location, because the NSP form's action-attribute specifies only the name of a targeted server page. If the NSP form contains submit buttons with own callees, the form is mapped to a JSP form that targets a specific dispatching front component. An NSP submit button is mapped to an HTML input element of submit-type. The callee of a submit button is mapped to a name/value pair of the corresponding HTML input element, whereas the

NSP reserved word "callee" is chosen as name. The name/value pair of an HTML submit button is only transmitted, if the form has been submitted with this button. Therefore the receiving dispatching front component can use the "callee"/callee pair to forward the form's request to the correct web server page. *Transforming NSP Controls and Web Signatures.* An NSP direct input control is mapped to a JSP input control with the type-attribute set to text. If a formal parameter of basic type is targeted, the mapping of the param-attribute is straightforward, it just becomes the name-attribute. Handling arrays and complex form data is explained below. The input-element's content, which possibly specifies a default value, becomes the value-attribute. NSP direct input controls and single select controls are active: if data entry must be type checked or if data entry respective selection is required, a JavaScript function is generated that is assigned to the onblur-event handler of the respective JSP input control. If the input control looses its focus, this function checks if the requirements are fulfilled and possibly pushes an appropriate warning to the user. More importantly a JavaScript validation function is assigned to the onSubmit-event handler of each form. JavaScript for checking requirements will be generated as part of that validation function. On submit, if a constraint violation has occurred, all violations will be reported and the form is being prevented from being submitted.

The other NSP controls are mapped to JSP controls accordingly. The transformation is interesting with respect to the values of arbitrary type. In the NSP approach arbitrary objects can be passed virtually to the user agent and back to a dialogue method on submit. Values of primitive type pose no problem, they can just be passed as Strings, whereas they are converted twice. For objects of non-primitive type an object reference must be passed virtually across the web user agent. If the object that should be passed is serializable an appropriately deep copy must be produced first. Then an object reference to this copy must be virtually passed across the web. For this purpose a surrogate string key is generated for the object reference in quest, the reference is stored under this key in a hashtable. The key is passed as a usual textual hidden parameter, select menu item, radio button value or check box value. If the key is received by a dialogue method it is used to retrieve the actual object from the hashtable.

If an NSP control targets a formal parameter of basic type, its param-attribute is mapped to the name-attribute of the JSP control. If an NSP control targets a formal array parameter, its param-attribute is mapped to the name-attribute of the JSP control again. But this time an index is attached to the control's value behind a special sign. An exception to this is the NSP element for a multiple select. The contained option elements take over the role of controls and their values are treated in the way just described.

An NSP control may be used in order to provide data for an attribute of an object net, that is for an item in an actual parameter of form message type. Such a control occurs underneath at least one NSP object-element. All object-elements and controls that together provide an actual parameter determine a labeled finitely branching tree. Leafs are given by controls, labels are given by form message attribute names or indexed attribute names in case of array attributes. Each control in such a tree determines a path expression made of labels. This path expression is concatenated to the actual control's value behind a special sign. The HTML name attribute is the param-attribute of the outermost NSP object-element that contains the NSP control in quest.

A param tag of a web server page is mapped to a get method on the Servlet request object that retrieves the value stored under the formal parameter's name. In case of a formal parameter of basic type the getParameter-method is executed. If the retrieved value has primitive type or has been provided by a direct input widget it is parsed with respect to the formal parameter's type. If the retrieved value is a key that represents an object that has been passed virtually across the net, it is used to get this object from the respective hashtable. In case of a formal array parameter a getParameterValues-method is executed and the index information is taken from the retrieved values. Based on the obtained values and index information an array is constructed. Similarly in case of a formal parameter of form message type the getParameterValues-method is executed and the path expressions, which specify the locations of the items in a complex object net, are taken from the retrieved values. Based on the obtained values and construction information an object net is instantiated.

Transforming Hyperlinks, Server-Side Calls, and Higher Order Server Pages. An NSP hyperlink is mapped to a JSP hyperlink. The form's callee-attribute is mapped to the XHTML form's href-attribute. Again the the site URL must be added to the callee. Furthermore the actual parameters of the NSP hyperlink, which are given as hidden parameters, must be added to the action URL as name/value pairs in the special XHTML syntax based on question mark, ampersand and equal sign.

An NSP call to an include server page is mapped to the JSP include-action. The callee-attribute of the NSP call-element is combined with the site URL and becomes the page-attribute of the JSP include-action. An actual parameter object of the call is stored into a hashtable under the targeted parameter's name. It is retrieved by the targeted server page.

Formal parameters of page type are used inside NSP form, link, and call elements tags. These usages must be enclosed in JSP expression scripting elements in the images of these tags under the described transformation.

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.

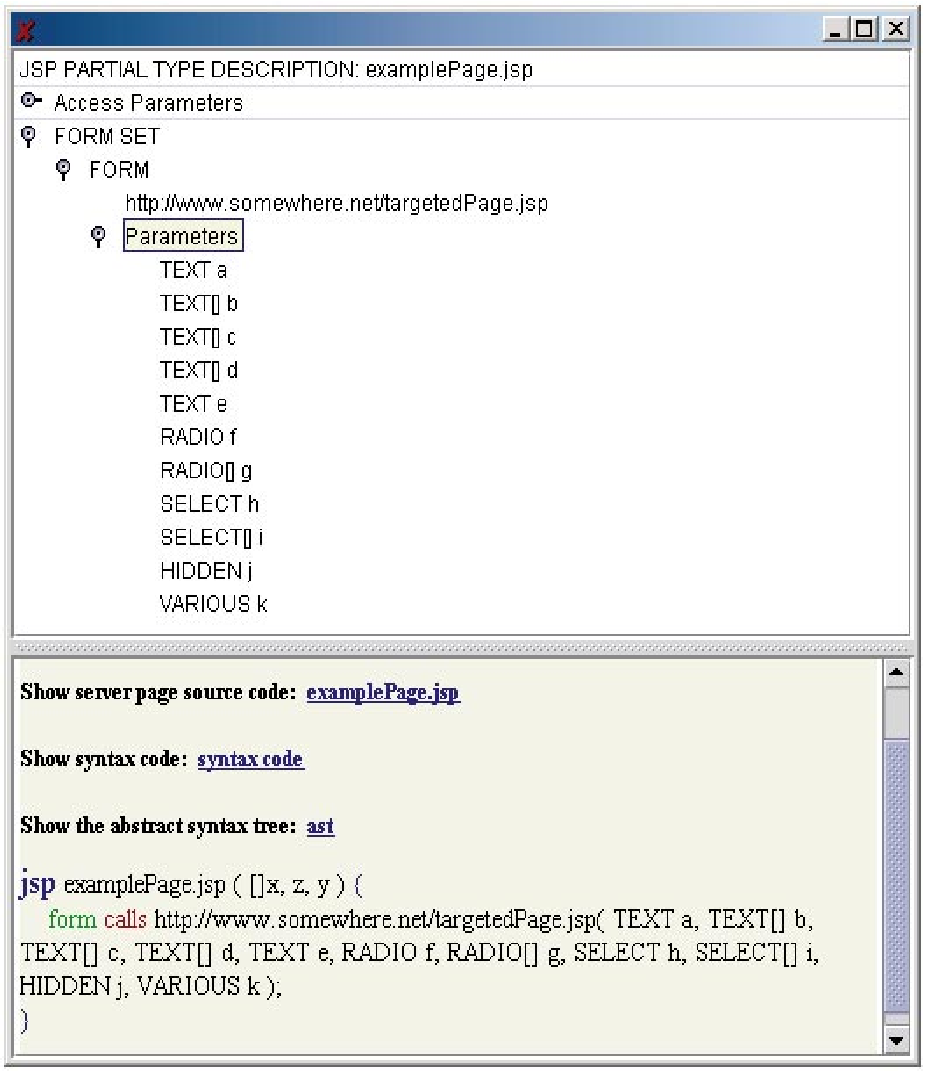
abstract syntax trees, type information, warnings and the linkage structure of the system in quest under several different viewpoints.

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.

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

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.

$$\begin{array}{rcl}
\underline{\texttt{pseudo}} & \texttt{::=} & (\underline{\texttt{element}})\texttt{+} \\
\\
\underline{\texttt{element}} & \texttt{::=} & \quad \underline{\texttt{control}} \\
 & & \texttt{|} \;\; \texttt{if } \underline{\texttt{pseudo}} \\
 & & \texttt{|} \;\; \texttt{ifElse } \underline{\texttt{pseudo}} \; \underline{\texttt{pseudo}} \\
 & & \texttt{|} \;\; \texttt{switch}^{\texttt{arbitrary}} \; (\underline{\texttt{pseudo}})\texttt{+} \\
 & & \texttt{|} \;\; \texttt{switch}^{\texttt{unique}} \; (\underline{\texttt{pseudo}})\texttt{+} \\
 & & \texttt{|} \;\; \texttt{loop } \underline{\texttt{pseudo}}
\end{array}$$

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.

$$\begin{array}{rcl}
\underline{\texttt{control}} & \texttt{::=} & \underline{\texttt{controltype}} \; \underline{\texttt{name}} \\
\\
\underline{\texttt{controltype}} & \texttt{::=} & \quad \texttt{text | textarea | password} \\
 & & \texttt{|} \;\; \texttt{hidden | checkbox | radiobutton} \\
 & & \texttt{|} \;\; \texttt{singleselect | multipleselect} \\
\\
\underline{\texttt{name}} & \texttt{::=} & n \in \boldsymbol{name}
\end{array}$$

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.

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

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.

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

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).

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

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.

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

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.

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

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.

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

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].

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

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

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

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

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

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

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.

Chapter 7

Formal Definition of the NSP Type System

This chapter formalizes the type system of Core NSP [63], which is the amalgamation of NSP concepts with a minimal imperative programming language similar to WHILE [71], which encompasses assignments, command sequences, a conditional control structure and an unbounded loop. The tag set of Core NSP consists of the most important elements for writing forms as well as some nestable text layout tags. The programming language types of Core NSP comprise records, arrays, and recursive types for modeling all the complexity found in the type system of a modern high-level programming languages. The web signatures of Core NSP embrace server page type parameters, that is higher order server pages are modeled. Tags for server side calls to server pages belong to the language, that is functional server page decomposition is modeled. A Per Martin-Löf [115][116][132] style type system is given to specify type correctness. The aims of this formal NSP type system definition are manifold:

- Preciseness. The NSP Coding guidelines and rules give an informal explanation of NSP type correctness. They are easy to learn and will help in everyday programming tasks, but may give rise to ambiguity. A precise description of the static semantics of NSP languages is desired. The formal Core NSP type system provides a succinct precise definition at the right level of communication.

- Justification. The implementations of the JSPick tool already convinces that NSP concepts like static client page description and type safety for server pages really work in practice. A formal type system definition makes it obvious.

- Linkage. A formal type system definition makes it easier to adapt results from the vast amount of literature on type systems to the NSP approach, especially concerning type inference resp. type checking algorithms.

The results of this chapter are summarized in appendix B in order to provide a comprehensive easy reference. Furthermore example type derivation are given in this appendix in B.7 and B.8.

7.1 Core NSP Grammar

An abstract syntax of Core NSP programs is specified by a context free grammar[1] [2]. In appendix C.1 an alternative specification of Core NSP as an XML document is given by a document type definition (DTD) [19]. A Core NSP program is a whole closed system of several server pages. A page is a parameterized core document and may be a complete web server page or an include server page:

$\underline{\texttt{system}}$	`::=`	$\underline{\texttt{page}}$ `|` $\underline{\texttt{system}}$ $\underline{\texttt{system}}$
$\underline{\texttt{page}}$	`::=`	`<nsp name="`$\underline{\texttt{id}}$`">` $\underline{\texttt{websig-core}}$ `</nsp>`
$\underline{\texttt{websig-core}}$	`::=`	$\underline{\texttt{param}}$ $\underline{\texttt{websig-core}}$ `|` $\underline{\texttt{webcall}}$ `|` $\underline{\texttt{include}}$
$\underline{\texttt{param}}$	`::=`	`<param name="`$\underline{\texttt{id}}$`" type="`$\underline{\texttt{parameter-type}}$`"/>`
$\underline{\texttt{webcall}}$	`::=`	`<html>` $\underline{\texttt{head}}$ $\underline{\texttt{body}}$ `</html>`
$\underline{\texttt{head}}$	`::=`	`<head><title>` $\underline{\texttt{strings}}$ `</title></head>`
$\underline{\texttt{strings}}$	`::=`	ε `|` $\underline{\texttt{string}}$ $\underline{\texttt{strings}}$
$\underline{\texttt{body}}$	`::=`	`<body>` $\underline{\texttt{dynamic}}$ `</body>`
$\underline{\texttt{include}}$	`::=`	`<include>` $\underline{\texttt{dynamic}}$ `</include>`

There are some basic syntactic categories. The category id is a set of labels. The category string consists of character strings[3]. The category parameter-type consists of the possible formal parameter types, i.e. programming language types plus page types. The category supported-type contains each type for which a direct manipulation input capability exists. The respective Core NSP types are specified in section 7.3.

$\underline{\texttt{string}}$	`::=`	$\mathrm{s} \in \mathbf{String}$
$\underline{\texttt{id}}$	`::=`	$\mathrm{l} \in \mathbf{Label}$
$\underline{\texttt{parameter-type}}$	`::=`	$\mathrm{t} \in \mathbb{T} \cup \mathbb{P}$
$\underline{\texttt{supported-type}}$	`::=`	$\mathrm{t} \in \mathbb{B}_{supported}$

Parameterized server pages are based on a dynamic markup language, which combines static client page description parts with active code parts. The static parts encompass lists, tables, server side calls, and forms with direct input capabilities, namely check boxes, select lists, and hidden parameters together with the object element for record construction.

[1]Nonterminals are underlined. Terminals are not emphasized - contrariwise to BNF standards like [97] or [46], however it fosters readability significantly.

[2]Every nonterminal corresponds to a syntactic category. In the grammar a syntactic category is depicted in bold face.

[3]A character string does not contain white spaces. We work with abstract syntax and therefore don't have to deal with white space handling problems [19].

$$\begin{array}{lcl}
\underline{\texttt{dynamic}} & ::= & \underline{\texttt{dynamic}}\ \underline{\texttt{dynamic}} \\
 & | & \varepsilon \mid \underline{\texttt{string}} \\
 & | & \underline{\texttt{ul}} \mid \underline{\texttt{li}} \\
 & | & \underline{\texttt{table}} \mid \underline{\texttt{tr}} \mid \underline{\texttt{td}} \\
 & | & \underline{\texttt{call}} \\
 & | & \underline{\texttt{form}} \mid \underline{\texttt{object}} \mid \underline{\texttt{hidden}} \mid \underline{\texttt{submit}} \\
 & | & \underline{\texttt{input}} \mid \underline{\texttt{checkbox}} \\
 & | & \underline{\texttt{select}} \mid \underline{\texttt{option}} \\
 & | & \underline{\texttt{expression}} \\
 & | & \underline{\texttt{code}}
\end{array}$$

Core NSP comprises list and table structures for document layout. All the XML elements of the dynamic markup language are direct subcategories of the category dynamic, which means that the grammar does not constrain arbitrary nesting of these elements. Instead of that the manner of use of a document fragment is maintained by the type systems. We delve on this in section 7.2.

$$\begin{array}{rcl}
\underline{\texttt{ul}} & ::= & \texttt{<ul>}\ \underline{\texttt{dynamic}}\ \texttt{</ul>} \\
\underline{\texttt{li}} & ::= & \texttt{<li>}\ \underline{\texttt{dynamic}}\ \texttt{</li>} \\
\underline{\texttt{table}} & ::= & \texttt{<table>}\ \underline{\texttt{dynamic}}\ \texttt{</table>} \\
\underline{\texttt{tr}} & ::= & \texttt{<tr>}\ \underline{\texttt{dynamic}}\ \texttt{</tr>} \\
\underline{\texttt{td}} & ::= & \texttt{<td>}\ \underline{\texttt{dynamic}}\ \texttt{</td>}
\end{array}$$

The rest of the static language parts address server side page calls, client side page calls and user interaction. A call may contain actual parameters only[4].

$$\begin{array}{rcl}
\underline{\texttt{call}} & ::= & \texttt{<call callee="}\underline{\texttt{id}}\texttt{">}\ \underline{\texttt{actualparams}}\ \texttt{</call>} \\
\underline{\texttt{actualparams}} & ::= & \varepsilon_{\texttt{act}} \mid \texttt{actualparam actualparams} \\
\underline{\texttt{actualparam}} & ::= & \texttt{<actualparam param="}\underline{\texttt{id}}\texttt{">}\ \underline{\texttt{expr}}\ \texttt{</actualparam>} \\
\underline{\texttt{form}} & ::= & \texttt{<form callee="}\underline{\texttt{id}}\texttt{">}\ \underline{\texttt{dynamic}}\ \texttt{</form>} \\
\underline{\texttt{object}} & ::= & \texttt{<object param="}\underline{\texttt{id}}\texttt{">}\ \underline{\texttt{dynamic}}\ \texttt{</object>} \\
\underline{\texttt{hidden}} & ::= & \texttt{<hidden param="}\underline{\texttt{id}}\texttt{">}\ \underline{\texttt{expr}}\ \texttt{</hidden>} \\
\underline{\texttt{submit}} & ::= & \texttt{<submit/>} \\
\underline{\texttt{input}} & ::= & \texttt{<input type="}\underline{\texttt{supported-type}}\texttt{" param="}\underline{\texttt{id}}\texttt{"/>} \\
\underline{\texttt{checkbox}} & ::= & \texttt{<checkbox param="}\underline{\texttt{id}}\texttt{"/>} \\
\underline{\texttt{select}} & ::= & \texttt{<select param="}\underline{\texttt{id}}\texttt{">}\ \underline{\texttt{dynamic}}\ \texttt{</select>} \\
\underline{\texttt{option}} & ::= & \texttt{<option>} \\
 & & \quad\texttt{<value>}\ \underline{\texttt{expr}}\ \texttt{</value>} \\
 & & \quad\texttt{<label>}\ \underline{\texttt{expr}}\ \texttt{</label>} \\
 & & \texttt{</option>}
\end{array}$$

Core NSP comprises expression tags for direct writing to the output and code

[4]The call element may contain no element, too. As a matter of taste the special sign $\varepsilon_{\texttt{act}}$ for empty contents is used in the Core NSP grammar to avoid redundant production and typing rules for the call element.

tags in order to express the integration of active code parts with layout[5].

```
expression  ::=  <expression> expr </expression>
      code  ::=  <code> com </code>
       com  ::=  </code> dynamic <code>
```

The imperative sublanguage of Core NSP comprises statements, command sequences, an if-then-else construct and a while loop.

```
com  ::=     stat
          |  com ; com
          |  if expr then com else com
          |  while expr do com
```

The only statement is assignment. Expressions are just variable values or deconstructions of complex variable values, i.e. arrays or user defined typed objects.

```
stat  ::=  id := expr
expr  ::=  id  | expr.id | expr[expr]
```

Core NSP is not a working programming language. It possesses only a set of most interesting features to model all the complexity of NSP technologies[6].

Instead Core NSP aims to specify the typed interplay of server pages, the interplay of static and active server page parts and the non-trivial interplay of the several complex types, i.e. user defined types and arrays, which arise during dynamically generating user interface descriptions.

7.2 Core NSP Type System Strength

The grammar given in 7.1 does not prevent arbitrary nestings of the several Core NSP dynamic tag elements. Instead necessary constraints on nesting are guaranteed by the type system. Therefore the type of a server page fragment comprises information about the manner of use of itself as part of an encompassing document.

As a result some context free properties are dealt with in static semantics. There are pragmatic reasons for this. Consider an obvious examples first. In HTML forms must not contain other forms. Furthermore some elements like the ones for input capabilities may only occur inside a form. If one wants to take such constraints into account in a context free grammar, one must create a non-terminal for document fragments inside forms and duplicate and appropriately

[5]The possibility to integrate layout code into active parts is needed. It is given by reversing the code tags. This way all Core NSP programs can be easily related to a convenient concrete syntax.

[6]Core NSP is even not turing-complete. Build-in operations, i.e. a sufficient powerful expression language, must be added. But this would just result in a more complex type system without providing more insight into static NSP semantics.

modify all the relevant production rules found so far. If there exist several such constraints the resulting grammar would quickly become unmaintainable. For that reason the Standard Generalized Markup Language supports the notions of exclusion and inclusion exception. The declaration of the HTML form element in the HTML 2.0 SGML DTD [40] is the following:

```
<!ELEMENT FORM - - %body.content -(FORM) +(INPUT|SELECT|TEXTAREA)>
```

The expression `-(FORM)` uses exclusion exception notation and the expression `+(INPUT|SELECT|TEXTAREA)` uses inclusion exception notation exactly for establishing the mentioned constraints. Indeed the SGML exception notation does not add to the expressive power of SGML [184], because an SGML expression that includes exceptions can be translated into an extended context free grammar [103][7]. The transformation algorithm given in [103] produces $2^{2|\mathtt{N}|}$ nonterminals in the worst case. This shows: if one does not have the exception notation at hand then one needs another way to manage complexity. The Core NSP way is to integrate necessary information into types, the resulting mechanism formalizes the way the Amsterdam SGML parser [181] handles exceptions[8].

Furthermore in NSP the syntax of the static parts is orthogonal to the syntax of the active parts, nevertheless both syntactic structures must regard each other. For example HTML or XHTML lists must not contain other elements than list items. The corresponding SGML DTD [98] and XHTML DTD [173] specifications are:

```
<!ELEMENT (OL|UL) - - (LI)+> resp.  <!ELEMENT ul (li)+>
```

In Core NSP the document fragment in listing 7.1 is considered correct.

Listing 7.1

```
<ul>
  <code> x:=3; </code>
  <li>First list item</li>
  <code>
    if condition then </code>
      <li>Second list item</li> <code>
    else </code>
      <li>Second list item</li> <code>
  </code>
</ul>
```

[7]An extended context free grammar is a context free grammar with production rules that may have regular expressions as right hand sides.

[8]The Amsterdam SGML parser deals with exceptions by keeping track of excluded elements in a stack.

Line 2 must be ignored with respect to the correct list structure, furthermore it must be recognized that the code in lines 4 to 11 corretly provides a list item. Again excluding wrong documents already by abstract syntax amounts to duplicate production rules for the static parts that may be contained in dynamic parts.

A Core NSP type checker has to verify uniquely naming of server pages in a complete system, which is a context dependent property. It has to check whether include pages provide correct elements. The way Core NSP treats dynamic fragment types fits seamlessly to these tasks.

Related Work

WASH/HTML is a mature embedded domain specific language for dynamic XML coding in the functional programming language Haskell, which is given by combinator libraries [168][169]. In [169] four levels of XML validity are defined. Well-formedness is the property of correct block structure, i.e. correct matching of opening and closing tags. Weak validity and elementary validity are both certain limited conformances to a given document type definition (DTD). Full validity is full conformance to a given DTD. The WASH/HTML approach can guarantee full validity of generated XML. It only guarantees weak validity with respect to the HTML SGML DTD under an immediate understanding of the defined XML validity levels for SGML documents[9]. As a reason for this the exclusion and inclusion exceptions in the HTML SGML DTD are given. The problem is considered less severe for the reason that in the XHTML DTD [173] exceptions only occur as comments[10] and XHTML has been created to overcome HTML. Unfortunately these comments become normative status in the corresponding XHMTL standard [172]; they are called element prohibitions. Therefore the problem of weak versus full validity remains an issue for XHTML, too.

The Core NSP type system shows that it is possible to statically ensure normative element prohibitions of the XHMTL standard. Anyway, despite questions concerning concrete technologies like fulfilling HTML/XHMTL are very interesting, the NSP approach targets to understand user interface description safety on a more conceptual level: the obvious, nonetheless important, statement is that it is possible to check arbitrary context free constraints on tag element nestings[11].

[9]There are a couple of other projects for dynamic XML generation, that guarantee some level of user interface description language safety, e.g. [78][86][120]. We delve on some further representative examples. In [180] two approaches are investigated. The first provides a library for XML processing arbitrary documents, thereby ensuring well-formedness. The second is a type-based translation framework for XML documents with respect to a given DTD, which guarantees full XML validity. Haskell Server Pages [122] guarantee well-formedness of XML documents. The small functional programming language XMλ [162] is based on XML documents as basic datatypes and is designed to ensure full XML validity [121].

[10]In XML DTDs no exception mechanism is available.

[11]In [157][17] it is shown that the normative element prohibitions of the XHMTL standard [172] can be statically checked by employing flow analysis [131][140][139].

7.3 Core NSP Types

In this section the types of Core NSP and the subtype relation between types are introduced simultaneously.

- Core NSP types. There are types for modeling programming language types, and special types for server pages and server page fragments in order to formalize the NSP coding guidelines and rules. The Core NSP types are given by a family of recursively defined type sets. Some of the Z mathematical toolkit notation [163] is used. Every type represents an infinite labeled regular tree.

- Core NSP subtyping. The subtype relation formalizes the relationship of actual client page parameters and formal server page parameters by strictly applying the Barbara Liskov principle [111][12]. A type A is subtype of another type B if every actual parameter of type A may be used in server page contexts requiring elements of type B. The subtype relation is defined as the greatest fix point of a generating function. The generating function is presented by a set of convenient judgment rules for deriving judgments of the form $\vdash\ S < T$.

7.3.1 Programming Language Types

In order to model the complexity of current high-level programming language type systems, the Core NSP types comprise basic types $\mathbb{B}_{primitive}$ and $\mathbb{B}_{supported}$, array types $\mathbb{A}$, record types $\mathbb{R}$, and recursive types $\mathbb{Y}$. $\mathbb{B}_{primitive}$ models types, for which no null object is provided automatically on submit. $\mathbb{B}_{supported}$ models types, for which a direct manipulation input capability exists. Note that $\mathbb{B}_{primitive}$ and $\mathbb{B}_{supported}$ overlap because of the int type. The set of all basic types $\mathbb{B}$ is made of the union of $\mathbb{B}_{primitive}$ and $\mathbb{B}_{supported}$. Record types and recursive types play the role of user defined form message types. The recursive types allow for modeling cyclic user defined data types. Thereby Core NSP works solely with structural type equivalence [26], i.e. there is no concept of introducing named user defined types, which would not contribute to the understanding of NSP concepts[13]. The types introduced so far and the type variables $\mathbb{V}$ together form the set of programming language types $\mathbb{T}$.

$$\mathbb{T} = \mathbb{B} \cup \mathbb{V} \cup \mathbb{A} \cup \mathbb{R} \cup \mathbb{Y}$$

[12]Liskov Substitution Principle: If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T.

[13]Enumeration types $\mathbb{E}$, i.e subtype polymorphism in the narrow sense of [26], could be introduced in order to specify the behavior of radio button tag structures.

$$\begin{array}{lcl}
\mathbb{B} & = & \mathbb{B}_{primitive} \cup \mathbb{B}_{supported} \\[1em]
\mathbb{B}_{primitive} & = & \{\texttt{int}, \texttt{float}, \texttt{boolean}\} \\[1em]
\mathbb{B}_{supported} & = & \{\texttt{int}, \texttt{Integer}, \texttt{String}\} \\[1em]
\mathbb{V} & = & \{X, Y, Z, \ldots\} \\
 & \cup & \{\texttt{Person}, \texttt{Customer}, \texttt{Article}, \ldots\}
\end{array}$$

Type variables may be bound by the recursive type constructor μ. Overall free type variables, that is type variables free in an entire Core NSP system resp. complete Core NSP program, represent opaque object reference types[14].

For every programming language type, there is an array type. According to subtyping rule 7.1 every type is subtype of its immediate array type. In commonly typed programming languages it is not possible to use a value as an array of the value's type. But the Core NSP subtype relation formalizes the relationship between actual client page and formal server page parameters. It is used in the NSP typing rules very targeted to constrain data submission. A single value may be used as an array if it is submitted to a server page.

In due course we informally distinguish between establishing subtyping rules and preserving subtyping rules. The establishing subtyping rules introduce initial NSP specific subtypings. The preserving subtyping rules are just the common judgments that deal with defining the effects of the various type constructors on the subtype relation. Judgment rule 7.2 is the preserving subtyping rule for array types.

$$\mathbb{A} = \{\ \texttt{array of}\ T \mid T \in \mathbb{T} \setminus \mathbb{A}\}$$

$$\frac{}{\vdash T < \texttt{array of}\ T} \tag{7.1}$$

$$\frac{\vdash S < T}{\vdash \texttt{array of}\ S < \texttt{array of}\ T} \tag{7.2}$$

A record is a finite collection of uniquely labeled data items, its fields. A record type is a finite collection of uniquely labeled types. In [145] record types are deliberately introduced as purely syntactical and therefore ordered entities. Then permutation rules are introduced that allow record types to be equal up to ordering. In other texts, like e.g. [2] or [158], record types are considered unordered from the beginning. We take the latter approach: a record type is a function from a finite set of labels to the set of programming language types. The usage of some Z Notation[15] [185][91] will ease writing type operator definitions and

[14]It is a usual economy not to introduce ground types in the presence of type variables [27]. Similarly in Core NSP example programs free term variables are used to model basic constant data values.

[15]$A \mathrel{+\!\!\!+\!\!\!\!\rightarrow} B$ is the set of all finite partial functions from A to B.

typing rules later on.

$$\mathbb{R} = \textbf{Label} \mathrel{+\!\!\!\!\!\!\mapsto} \mathbb{T}$$

$$\frac{T_j \notin \mathbb{B}_{primitive} \quad j \in 1 \ldots n}{\vdash \{l_i \mapsto T_i\}^{i \in 1 \ldots j-1, j+1 \ldots n} < \{l_i \mapsto T_i\}^{i \in 1 \ldots n}} \tag{7.3}$$

$$\frac{\vdash S_1 < T_1 \ldots \vdash S_n < T_n}{\vdash \{l_i \mapsto S_i\}^{i \in 1 \ldots n} < \{l_i \mapsto T_i\}^{i \in 1 \ldots n}} \tag{7.4}$$

Rule 7.4 is just the necessary preserving subtyping rule for records.

The establishing subtyping rule 7.3 states that a shorter record type is subtype of a longer record type, provided the types are equal with respect to labeled type variables. At a first site this contradicts the well-known rules for subtyping records [27] or objects [1]. But there is no contradiction, because these rules describe hierarchies of feature support[16] and we just specify another phenomenon: rule 7.3 models that an actual record parameter is automatically filled with null objects for the fields of non-primitive types that are not provided by the actual parameter, but expected by the formal parameter.

The Core NSP type system encompasses recursive types for modeling the complexity of cyclic user defined data types.

$$\mathbb{Y} = \{ \; \mu \, X \; . \; R \mid X \in \mathbf{V} \; , \; R \in \mathbb{R} \; \}$$

$$\frac{\vdash S[{}^{\mu X.S}/_{X}] < T}{\vdash \mu X.S < T} \tag{7.5}$$

$$\frac{\vdash S < T[{}^{\mu X.T}/_{X}]}{\vdash S < \mu X.T} \tag{7.6}$$

Recursive types may be handled in an iso-recursive or an equi-recursive way[17]. In an iso-recursive type system, a recursive type is considered isomorphic to its one-step unfolding and a family of unfold and fold operations on the term level is provided in order to represent the type isomorphisms. A prominent example of this purely syntactical approach is [2]. In an equi-recursive type system like the one given in [10], two recursive types are considered equal if they have the same infinite unfolding. We have chosen to follow the equi-recursive approach along the lines of [76] for two reasons. First it keeps the Core NSP language natural, no explicit fold and unfolding is needed. More importantly, though the theory of an equi-recursive treatment is challenging[18], it is well-understood and some crucial results concerning proof-techniques and type checking of recursive typing

[16]In [55] object subtyping is explained strictly along the notion of feature support.

[17]The terms iso-recursive or an equi-recursive stem from [45].

[18]Compared to iso-recursive typing, an equi-recursive typing is a Curry-style [10], i.e implicit typing discipline, and therefore its theory is more challenging, or in more practical terms, constructing a type checker is more challenging.

and recursive subtyping like [28][3][160][101] are elaborated in an equi-recursive setting.

The subtype relation adequately formalizes all the advanced NSP notions like form message types and higher order server pages as may be checked against the examples given in chapter 3 and the sample type derivations in appendices B.7 and B.8.

In the Core NSP type system types represent finite trees or possibly infinite regular trees[19][42]. More precisely these type trees are unordered[20], labeled, and finitely branching. Type equivalence is not explicitly defined, it is given implicitly by the subtype relation: the subtype relation is not a partial order but a pre-order and two types are equal if they are mutual subtypes. The subtype relation is defined in this section as the greatest fixpoint of a monotone generating function on the universe of type trees [76]. The Core NSP subtyping rules provide an intuitive description of this generating function. Thereby the subtyping rules for left folding 7.5 and right folding 7.6 provide the desired recursive subtyping.

Beyond this only one further subtyping rule is needed, namely the rule 7.7 for introducing reflexivity. No explicit introduction of transitivity must be provided as in iso-recursive type systems, because this property already follows from the definition of the subtype relation as greatest fixed point of a generating function [76].

$$\frac{}{\vdash T < T} \tag{7.7}$$

We will continue to explain server page types in section 7.3.2. But first, a digression on Core NSP recursive record subtyping.

A Digression on Core NSP Recursive Record Subtyping

Normally sum or variant types are used in order to provide some finite data to be inhabited in recursive types. There is no sum or variant type in Core NSP. Instead the establishing record subtyping rule 7.3 makes it possible that finite data may be submitted to a formal parameter of recursive type. For example the finite data gathered in listing 7.2 has the recursive type given by subtype derivation 7.8.

$$e \equiv_{\texttt{DEF}} element$$
$$n \equiv_{\texttt{DEF}} next$$

$$\begin{array}{ll} \vdash & \{e \mapsto int, n \mapsto \{e \mapsto int, n \mapsto \{e \mapsto int\}\}\} \\ < & \{e \mapsto int, n \mapsto \{e \mapsto int, n \mapsto \{e \mapsto int, n \mapsto \mu X.\{e \mapsto int, n \mapsto X\}\}\}\} \\ = & \mu X.\{e \mapsto int, n \mapsto X\} \end{array} \tag{7.8}$$

[19]A regular tree is a possibly infinite tree that has a finite set of distinct subtrees only.
[20]We treat records as unordered tuples from the outset.

Listing 7.2

```
<object param="list">
  <input param="element" type="int"/>
  <object param="next">
    <input param="element" type="int"/>
    <object param="next">
      <input param="next" type="int"/>
    </object>
  </object>
</object>
```

Interestingly the record subtyping rule 7.3 is more generally appropriate for a direct formalization of cyclic data type definitions found in usual object-oriented languages. Consider the definition of possibly infinite lists in the Haskell [144] code fragment 7.9. The null data constructor for the left summand is needed in order to explicitly enable finite lists.

$$\texttt{data}\ \ \texttt{List} = \texttt{NULL} \ \mid\ \texttt{NODE}\{\texttt{element} :: \texttt{Int}, \texttt{next} :: \texttt{List}\} \tag{7.9}$$

The direct counterpart of this data type definition in an object-oriented language is depicted as an UML diagram[21] in Figure 7.1(a). A more appropriate way of modeling lists in an object-oriented programming language is given in Figure 7.1(b). Basically the 1 multiplicity of the navigated association in 7.1(a) is replaced by an 0..1 multiplicity in 7.1(b). It is possible to define lists this way, because in an object-oriented language every object field of complex type that is not constructed is implicitly provided as a null object. This mechanism is exactly the one formalized by subtyping rule 7.3. To see this, note that the lists defined in equation 7.9 and 7.1(a) both have type $\texttt{List}_{\texttt{expl}}$ given in 7.10[22], whereas 7.1(b) has type $\texttt{List}_{\texttt{impl}}$ given in equation 7.11 provided that subtyping rule 7.3 is present.

$$\texttt{List}_{\texttt{expl}} \equiv_{\texttt{DEF}} \mu List.[\texttt{NULL} : \{\}, \texttt{NODE} : \{element \mapsto \texttt{int}, next \mapsto List\}] \tag{7.10}$$

$$\texttt{List}_{\texttt{impl}} \equiv_{\texttt{DEF}} \mu List.\{element \mapsto \texttt{int}, next \mapsto List\} \tag{7.11}$$

7.3.2 Server Page Types

In order to formalize the NSP coding guidelines and rules the type system of Core NSP comprises server page types $\mathbb{P}$, web signatures $\mathbb{W}$, a single complete web page type $\square \in \mathbb{C}$, dynamic fragment types $\mathbb{D}$, layout types $\mathbb{L}$, tag element types $\mathbb{E}$, form occurrence types $\mathbb{F}$ and system types $\mathbb{S}$.

[21] The diagram is drawn within the an implementation perspective in the sense of [74][41].

[22] The variant notation is taken from [83].

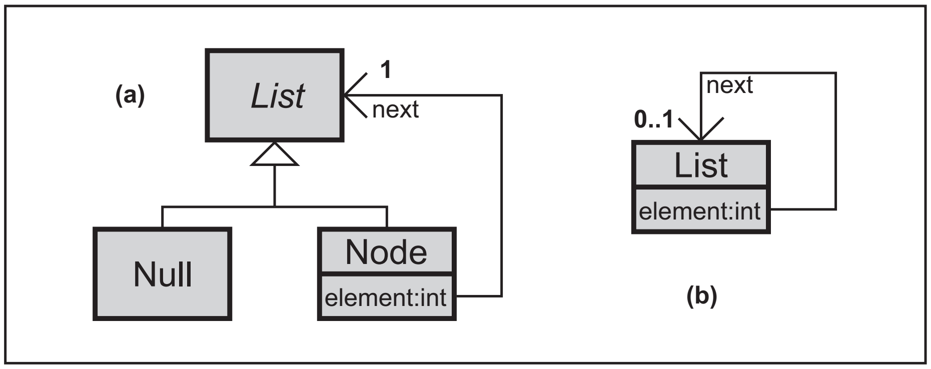


Figure 7.1: UML list definitions. The figure shows to alternative cyclic data models of the list data type.

A server page type is a functional type, that has a web signature as argument type. An include server page has a dynamic document fragment type as result type, and a web server page the unique complete web page type.

$$\mathbb{P} = \{\ w \rightarrow r \mid w \in \mathbb{W}\ ,\ r \in \mathbb{C} \cup \mathbb{D}\ \}$$

$$\mathbb{W} = \textbf{Label} \twoheadrightarrow\!\!\!\!\!\!\not\;\;\; (\mathbb{T} \cup \mathbb{P})$$

$$\mathbb{C} = \{\Box\}$$

A web signature is a record. This time a labeled component of a record type is either a programming language type or a server page type, that is the type system supports higher order server pages. Noteworthy a clean separation between the programming language types and the additional NSP specific types is kept. Server page types may be formal parameter types, but these formal parameters can be used only by specific NSP tags. Server pages deliberately become no first class citizens, because this way the Core NSP models conservative amalgamation of NSP concepts with a high-level programming language.

The preserving subtyping rule 7.4 for records equally applies to web signatures. The establishing subtyping rule 7.3 must be slightly modified resulting in rule 7.12, because formal parameters of server page type must always be provided, too.

Subtyping rule 7.13 is standard and states, that server page types are contravariant in the argument type and covariant in the result type.

$$\frac{T_j \notin \mathbb{B}_{primitive} \cup \mathbb{P} \quad j \in 1 \ldots n}{\vdash \{l_i \mapsto T_i\}^{i \in 1 \ldots j-1, j+1 \ldots n} < \{l_i \mapsto T_i\}^{i \in 1 \ldots n}} \tag{7.12}$$

$$\frac{\vdash w' < w \qquad \vdash R < R'}{\vdash w \rightarrow R < w' \rightarrow R'} \tag{7.13}$$

A part of a core document has a document fragment type. Such a type consists of a layout type and a web signature. The web signature is the type of the data, which is eventually provided by the document fragment as part of an actual form parameter. If a web signature plays part of a document fragment type it is also called form type. The layout type constrains the usability of the document fragment as part of an encompassing document. It consists of an element type and a form occurence type.

$$\mathbb{D} = \mathbb{L} \times \mathbb{W}$$

$$\mathbb{L} = \mathbb{E} \times \mathbb{F}$$

$$\frac{\vdash S_1 < T_1 \quad \vdash S_2 < T_2}{\vdash (S_1, S_2) < (T_1, T_2)} \tag{7.14}$$

Subtyping rule 7.14 is standard for products and applies both to layout and tag element types. An element type partly describes where a document fragment may be used. Document fragment that are sure to produce no output have the neutral document type $\circ$. Examples for such neutral document parts are hidden parameters and pure java code. Document fragments that may produce visible data like String data or controls have the output type $\bullet$. Document fragments that may produce list elements, table data, table rows or select list options have type **LI**,**TD**, **TR** and **OP**. They may be used in contexts where the respective element is demanded. Neutral code can be used everywhere. This is expressed by rule 7.15.

$$\mathbb{E} = \{\ \circ, \bullet, \mathbf{TR}, \mathbf{TD}, \mathbf{LI}, \mathbf{OP}\}$$

$$\frac{T \in \mathbb{E}}{\vdash \circ\ <\ T} \tag{7.15}$$

The form occurrence types further constrains the usability of document fragments. Fragments that must be used inside a form, because they generate client page parts containing controls, have the inside form type $\Downarrow$. Fragments that must be used outside a form, because they generate client page fragments that already contain forms, have the outside form type $\Uparrow$. Fragments that may be used inside or outside forms have the neutral form type $\Updownarrow$. Rule 7.16 specifies, that such fragments can play the role of both fragments of outside form and fragments of inside form type.

$$\mathbb{F} = \{\ \Downarrow, \Uparrow, \Updownarrow\ \}$$

$$\frac{T \in \mathbb{F}}{\vdash \Updownarrow <\ T} \tag{7.16}$$

$$\mathbb{S} = \{\ \diamond, \surd\ \}$$

An NSP system is a collection of NSP server pages. NSP systems that are type correct receive the well type $\diamond$. The complete type $\surd$ is used for complete systems. A complete system is a well typed system where all used server page names are defined, i.e. are assigned to a server page of the system, and no server page names are used as variables.

7.4 Type Operators

In the NSP typing rules in chapter 7.6 a central type operation, termed form type composition $\odot$ in the sequel, is used that describes the composition of form content fragments with respect to the provided actual superparameter type. First an auxiliary operator $*$ is defined, which provides the dual effect of the array item type extractor $\Downarrow$ in section 6.2. If applied to an array the operater lets the type unchanged, otherwise it yields the respective array type.

$$T* \equiv_{\texttt{DEF}} \begin{cases} \texttt{array of } T & , T \notin \mathbb{A} \\ T & , \mathit{else} \end{cases}$$

The form type composition $\odot$ is the corner stone of the NSP type system. Form content provides direct input capabilities, data selection capabilities and hidden parameters. On submit an actual superparameter is transmitted. The type of this superparameter can be determined statically in NSP, it is called the form type[23] of the form content. Equally document fragments, which dynamically may generate form content, have a form type. Form type composition is applied to form parameter types and describes the effect of sequencing document parts. Consequently form type composition is used to specify typing with respect to programming language sequencing, loops and document composition.

$$w_1 \odot w_2 \equiv_{\texttt{DEF}}$$

$$\begin{cases} \perp & , \boldsymbol{if}\ \exists (l_1 \mapsto T_1) \in w_1 \bullet\ \exists (l_2 \mapsto T_2) \in w_2 \bullet\ l_1 = l_2 \wedge P_1 \in \mathbb{P} \wedge P_2 \in \mathbb{P} \\ \perp & , \boldsymbol{if}\ \exists (l_1 \mapsto T_1) \in w_1 \bullet\ \exists (l_2 \mapsto T_2) \in w_2 \bullet\ l_1 = l_2 \wedge T_1 \sqcup T_2\ \mathit{undefined} \\ \begin{array}{l} \\ (\texttt{dom}\ w_2) \lessdot w_1 \cup (\texttt{dom}\ w_1) \lessdot w_2 \end{array} & \\ \cup\ \left\{ \left(l \mapsto (T_1 \sqcup T_2)*\right) | (l \mapsto T_1) \in w_1 \wedge (l \mapsto T_2) \in w_2 \right\} & , \boldsymbol{else} \end{cases}$$

If a document fragment targets a formal parameter of a certain type and another document fragment does not target this formal parameter, then and only then the document resulting from sequencing the document parts targets the given formal parameter with unchanged type. That is, with respect to non-overlapping parts of form types, form type composition is just union. With antidomain restriction notation[24] this is specified succinctly in line 3 of the $\odot$ operator definition.

[23] section 7.3.2
[24] Appendix D

Two document fragments that target the same formal parameters may be sequenced, if the targeted formal parameter types are compatible for each formal parameter. NSP types are compatible if they have a supertype in common. The NSP subtype relation formalizes when an actual parameter may be submitted to a dialogue method: if its type is a subtype of the targeted formal parameter. So if two documents have targeted parameters with compatible types in common only, the joined document may target every dialogue method that fulfills the following: formal parameters that are targeted by both document parts have an array type, because of sequencing a single data transmission cannot be ensured in neither case, thereby the array items' type must be a common supertype of the targeting actual parameters. This is formalized in line 4 of the the $\odot$ operator definition: for every shared formal parameter a formal array parameter of the least common supertype belongs to the result form type[25]. Consider the following example application of the $\odot$ operator:

$$\begin{array}{ll}
 & \{l \mapsto \texttt{int}, n \mapsto \{o \mapsto \texttt{int}, p \mapsto \texttt{String}\}\} \qquad (T_1) \\
\odot & \{m \mapsto \texttt{int}, n \mapsto \{o \mapsto \texttt{int}, q \mapsto \texttt{String}\}\} \qquad (T_2)
\end{array}$$

$$= \begin{array}{l}
\{ \quad l \mapsto \texttt{int}, \\
\quad\quad m \mapsto \texttt{int}, \\
\quad\quad n \mapsto \texttt{array of}\{o \mapsto \texttt{int}, p \mapsto \texttt{String}, q \mapsto \texttt{String}\} \\
\}
\end{array} \qquad (T_3)$$

In the example two form fragments are concatenated, the first one having type T_1, the second one having type T_2. The compound form content will provide int values for the formal parameters l and m. It will provide to actual parameters for the formal parameter n. Thereby the record stemming from the first form fragment can be automatically filled with a null object for a formal q parameter of type String, because String is a non-primitive type. Analogously, the record stemming from the second form fragment can be automatically filled with a null object for a formal p parameter. The compound form document therefore can target a dialogue method with web signature T_3. A complex example for the application of the $\odot$ operator in the interplay of subtyping and recursive typing can be found in appendix B.8.

The error cases in the $\odot$ operator definition are equally important. The $\odot$ operator is a partial function. If two document fragments target a same formal parameter with non-compatible types, they simply cannot be sequenced. The $\odot$ operator is undefined for the respective form types. More interestingly, two document fragments that should be composed must not target a formal server page parameter. This would result in an actual server page parameter array which would contradict the overall principle of conservative language amalgamation[26] introduced in chapter 1.

[25]The least common supertype of two types is given as least upper bound of the two types, which is unique up to the equality induced by recursive subtyping itself.

[26]If desired page array types must be introduced by tag support.

Form type composition can be characterized algebraically. The web signatures form a monoid ($\mathbb{W}$, $\odot$, $\emptyset$) with the $\odot$ operator as monoid operation and the empty web signature as neutral element. The operation $(\lambda v.v \odot w)_w$ is idempotent for every arbitrary fixed web signature w, which explains why the typing rule 7.28 for loop-structures is adequate.

7.5 Environments and Judgments

In the NSP type system two environments are used. The first environment Γ is the usual type environment. The second environment Δ is used for binding names to server pages, i.e. as a definition environment. It follows from their declaration that environments are web signatures. All definitions coined for web signatures immediately apply to the environments. This is exploited for example in the system parts typing rule 7.50.

$$\Gamma : \mathbf{Label} \twoheadrightarrow\!\!\!\!\!\!\mid\;\, (\mathbb{T} \cup \mathbb{P}) \quad = \mathbb{W}$$

$$\Delta : \mathbf{Label} \twoheadrightarrow\!\!\!\!\!\!\mid\;\, \mathbb{P} \qquad\qquad \subset \mathbb{W}$$

The Core NSP identifiers are used for basic programming language expressions, namely variables and constants, and for page identifiers, namely formal page parameters and server pages names belonging to the complete system. In some contexts, e.g. in hidden parameters or in select menu option values, both page identifiers and arbitrary programming language expressions are allowed. Therefore initially page identifiers are treated syntactically as programming language expressions. However a clean cut between page identifiers and the programming language is maintained, because the modeling of conservative amalgamation is an objective. The cut is provided by the premises of typing rules concerning such elements where only a certain kind of entity is allowed; e.g. in the statement typing rule 7.20 it is prevented that page identifiers may become program parts.

The Core NSP type system relies on several typing judgments:

$$\begin{array}{ll}
\Gamma \vdash e : \mathbb{T} \cup \mathbb{P} & e \in \mathbf{expr} \\
\Gamma \vdash n : \mathbb{D} & n \in \mathbf{com} \cup \mathbf{dynamic} \\
\Gamma \vdash c : \mathbb{P} & c \in \textbf{websig-core} \\
\Gamma \vdash a : \mathbb{W} & a \in \mathbf{actualparams} \\
\Gamma, \Delta \vdash s : \mathbb{S} & s \in \mathbf{system}
\end{array}$$

Eventually the judgment that a system has complete type is targeted. In order to achieve this, different kinds of types must be derived for entities of different syntactic categories. Expressions have programming language types or page types, consequently along the lines just discussed. Both programming language code and user interface descriptions have document fragment types, because they can be interlaced arbitrarily and therefore belong conceptually to the same kind of document. Parameterized core documents have page types. The actual parameters of a call element together provide an actual superparameter, the type of this is a web signature and is termed a call type. All the kinds of judgments so far work with respect to a given type environment. If documents are considered as parts of a system they must mutually respect defined server page names. Therefore subsystem judgments has to be given additionally with respect to the definition environment.

7.6 Typing Rules

The notion of Core NSP type correctness is specified as an algorithmic type system. In the presence of subtyping there are two alternatives for specifying type correctness with a type system. The first one is by means of a declarative type system. In such a type system a subsumption rule is present. Whenever necessary it can be derived that an entity has always each of its supertypes. Instead in an algorithmic type system reasoning about an entities' supertypes happens in a controlled way by fulfilling typing rule premises. Both approaches have their advantages and drawbacks. The declarative approach usually leads to more succinct typing rules whereas reasoning about type system properties may become complicated - cut elimination techniques may have to be employed. In the algorithmic approach the single typing rules may quickly become complex, however an algorithmic type system is easier to handle in proofs.

For Core NSP an algorithmic type system is the correct choice. Extra premises are needed in some of the typing rules, e.g. in the typing rule for form submission. In some rules slightly bit more complex type patterns have to be used in the premises, e.g. in the typing rules concerning layout structuring document elements. However in the Core NSP type system these extra complexity fosters understandability.

In this section the typing rules are presented by starting from basic building blocks to more complex building blocks. In appendix B.6 the rules are ordered with respect to the production rules in appendix B.1 for easy reference.

The typing rule 7.17 allows for extraction of an identifier typing assumption from the typing environment. Rules 7.18 and 7.19 give the types of selected record fields respectively indexed array elements.

$$\frac{(v \mapsto T) \in \Gamma}{\Gamma \vdash v : T} \tag{7.17}$$

$$\frac{\Gamma \vdash e : \{l_i \mapsto T_i\}^{i \in 1 \ldots n} \qquad j \in 1 \ldots n}{\Gamma \vdash e.l_j : T_j} \tag{7.18}$$

$$\frac{\Gamma \vdash e : \texttt{array of } T \qquad \Gamma \vdash i : \texttt{int}}{\Gamma \vdash e[i] : T} \tag{7.19}$$

Typing rule 7.20 introduces programming language statements, namely assignments. Only programming language variables and expression may be used, i.e. expressions must not contain page identifiers. The resulting statement is sure not to produce any output. It is possible to write an assignment inside forms and outside forms. If it is used inside a form it will not contribute to the submitted superparameter. Therefore a statement has a document fragment type which is composed out of the neutral document type, the neutral form type and the empty web signature. The empty string, which is explicitly allowed as content in NSP, obtains the same type by rule 7.21.

$$\frac{\Gamma \vdash x : T \qquad \Gamma \vdash e : T \qquad T \in \mathbb{T}}{\Gamma \vdash x := e \;:\; ((\circ, \Updownarrow), \emptyset)} \tag{7.20}$$

$$\frac{}{\Gamma \vdash \boldsymbol{\varepsilon} : ((\circ, \Updownarrow), \emptyset)} \tag{7.21}$$

Actually in Core NSP programming language and user interface description language are interlaced tightly by the abstract syntax. The code tags are just a means to relate the syntax to common concrete server pages syntax. The code tags are used to switch explicitly between programming language and user interface description and back. For the latter the tags may be read in reverse order. However this switching does not affect the document fragment type and therefore the rules 7.22 and 7.23 do not, too.

$$\frac{\Gamma \vdash c : D}{\Gamma \vdash < \texttt{code} > c < /\texttt{code} > \;:\; D} \tag{7.22}$$

$$\frac{\Gamma \vdash d : D}{\Gamma \vdash < /\texttt{code} > d < \texttt{code} > \;:\; D} \tag{7.23}$$

Equally basic as rule 7.20, rule 7.24 introduces character strings as well typed user interface descriptions. A string's type consists of the output type, the neutral form type and the empty web signature. Another way to produce output is by means of expression elements, which support all basic types and get by rule 7.25 the same type as character strings.

$$\frac{d \in \textbf{string}}{\Gamma \vdash d : ((\bullet, \Updownarrow), \emptyset)} \tag{7.24}$$

$$\frac{\Gamma \vdash e : T \quad T \in \mathbb{B}}{\Gamma \vdash < \texttt{expression} > e < /\texttt{expression} > \;:\; ((\bullet, \Updownarrow), \emptyset)} \tag{7.25}$$

Composing user descriptions parts and sequencing programming language parts must follow essentially the same typing rule. In both rule 7.26 and rule 7.27 premises ensure that the document fragment types of both document parts are compatible. If the parts have a common layout supertype, they may be used together in server pages contexts of that type. If in addition to that the composition of the parts' form types is defined, the composition becomes the resulting form type. Form composition has been explained in section 7.4.

$$\frac{\begin{array}{c} d_1, d_2 \in \textbf{dynamic} \\ \Gamma \vdash d_1 : (L_1, w_1) \quad \Gamma \vdash d_2 : (L_2, w_2) \quad L_1 \sqcup L_2 \downarrow \quad w_1 \odot w_2 \downarrow \end{array}}{\Gamma \vdash d_1 \ d_2 : (L_1 \sqcup L_2, w_1 \odot w_2)} \tag{7.26}$$

$$\frac{\Gamma \vdash c_1 : (L_1, w_1) \qquad \Gamma \vdash c_2 : (L_2, w_2) \qquad L_1 \sqcup L_2 \downarrow \qquad w_1 \odot w_2 \downarrow}{\Gamma \vdash c_1; c_2 \ : \ (L_1 \sqcup L_2, w_1 \odot w_2)} \tag{7.27}$$

The loop is a means of dynamically sequencing. From the type system's point of view it suffices to regard it as a sequence of twice the loop body as expressed by typing rule 7.28. For an if-then-else-structure the types of both branches must be compatible in order to yield a well-typed structure. Either one or the other branch is executed, so the least upper bound of the layout types and least upper bound of the form types establish the adequate new document fragment type.

$$\frac{\Gamma \vdash e : \texttt{boolean} \qquad \Gamma \vdash c : (L, w)}{\Gamma \vdash \texttt{while}\ e\ \texttt{do}\ c \ : \ (L, w \odot w)} \tag{7.28}$$

$$\frac{\Gamma \vdash e : \texttt{boolean} \quad \Gamma \vdash c_1 : D_1 \quad \Gamma \vdash c_2 : D_2 \qquad D_1 \sqcup D_2 \downarrow}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ c_1 \texttt{else}\ c_2 \ : \ D_1 \sqcup D_2} \tag{7.29}$$

Next the typing rules for controls are considered. The submit button is a visible control and must not occur outside a form, in Core NSP it is an empty element. It obtains the output type, the inside form type, and the empty web signature as document fragment type. Similarly an input control obtains the output type and the inside form type. But an input control introduces a form type. The type of the input control is syntactically fixed to be a widget supported type. The param-attribute of the control is mapped to the control's type. This pair becomes the form type in the control's document fragment type. Check boxes are similar. In Core NSP check boxes are only used to gather boolean data.

$$\frac{}{\Gamma \vdash < \texttt{submit}/ > \ : \ ((\bullet, \Downarrow), \emptyset)} \tag{7.30}$$

$$\frac{T \in \mathbb{B}_{supported}}{\Gamma \vdash \ < \texttt{input type} = "T" \ \texttt{param} = "l"/ > : ((\bullet, \Downarrow), \{(l \mapsto T)\})} \tag{7.31}$$

$$\frac{}{\Gamma \vdash <\texttt{checkbox}\quad\texttt{param} = "l"/> \;:\; ((\bullet, \Updownarrow), \{(l \mapsto boolean)\})} \tag{7.32}$$

Hidden parameters are not visible. They get the neutral form type as part of their fragment type. The value of the hidden parameter may be a programming language expression of arbitrary type or an identifier of page type.

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash <\texttt{hidden}\,\texttt{param} = "l"> e </\texttt{hidden}> \;:\; ((\circ, \Downarrow), \{(l \mapsto T)\})} \tag{7.33}$$

The select element may only contain code that generates option elements. Therefore an option element obtains the option type **OP** by rule 7.35 and the select element typing rule 7.34 requires this option type from its content. An option element has not an own param-element. The interesting type information concerning the option value is wrapped as an array type that is assigned to an arbitrary label. The type information is used by rule 7.34 to construct the correct form type[27].

$$\frac{\Gamma \vdash d : \big((\mathbf{OP}, \Updownarrow), \{(l \mapsto \texttt{array of } T)\}\big)}{\Gamma \vdash \begin{array}{l} <\texttt{select}\quad\texttt{param} = "l"> \\ \qquad d \\ </\texttt{select}> \;:\; ((\bullet, \Downarrow), \{(l \mapsto \texttt{array of } T)\}) \end{array}} \tag{7.34}$$

$$\frac{\Gamma \vdash v \;:\; T \qquad \Gamma \vdash e : S \quad S \in \mathbb{B} \qquad l \in \mathbf{Label}}{\Gamma \vdash \begin{array}{l} <\texttt{option}> \\ \qquad <\texttt{value}> v </\texttt{value}> \\ \qquad <\texttt{label}> e </\texttt{label}> \\ </\texttt{option}> : ((\mathbf{OP}, \Updownarrow), \{(l \mapsto \texttt{array of } T)\}) \end{array}} \tag{7.35}$$

The object element is a record construction facility. The enclosed document fragment's layout type lasts after application of typing rule 7.36, whereas the fragment's form type is assigned to the object element's param-attribute. This way the superparameter provided by the enclosed document becomes a named object attribute.

$$\frac{\Gamma \vdash d : (L, w)}{\Gamma \vdash <\texttt{object}\,\texttt{param} = "l"> d </\texttt{object}> \;:\; (L, \{(l \mapsto w)\})} \tag{7.36}$$

The form typing rule 7.37 requires that a form may target only a server page that yields a complete web page if it is called. Furthermore the form type of the form content must be a subtype of the targeted web signature, because the Core NSP subtype relations specifies when a form parameter may be submitted to a dialogue method of given signature. Furthermore the form content's must

[27]This way no new kind of judgment has to be introduced for select menu options.

be allowed to occur inside a form. Then the rule 7.37 specifies that the form is a vizible element that must not contain inside another form.

$$\frac{\Gamma \vdash l : w \rightarrow \Box \qquad \Gamma \vdash d : ((e, \Downarrow), v) \qquad \vdash v < w}{\Gamma \vdash <\texttt{form} \quad \texttt{callee} = "l" > d < /\texttt{form} > : ((e, \Uparrow), \emptyset)} \tag{7.37}$$

Now the layout structuring elements, i.e. lists and tables, are investigated. The corresponding typing rules 7.38 to 7.42 do not affect the form types and form occurrence types of contained elements. Only document parts that have no specific layout type, i.e. are either neutral or merely vizible, are allowed to become list items by rule 7.38. Only documents with list layout type may become part of a list. A well-typed list is a vizible element. The rules 7.40 to 7.42 work analogously for tables.

$$\frac{\Gamma \vdash d : ((\bullet \vee \circ, F), w)}{< \texttt{li} > d < /\texttt{li} > \ : \ ((\mathbf{LI}, F), w)} \tag{7.38}$$

$$\frac{\Gamma \vdash d : ((\mathbf{LI} \vee \circ, F), w)}{< \texttt{ul} > d < /\texttt{ul} > \ : \ ((\bullet, F), w)} \tag{7.39}$$

$$\frac{\Gamma \vdash d : ((\bullet \vee \circ, F), w)}{< \texttt{td} > d < /\texttt{td} > \ : \ ((\mathbf{TD}, F), w)} \tag{7.40}$$

$$\frac{\Gamma \vdash d : ((\mathbf{TD} \vee \circ, F), w)}{< \texttt{tr} > d < /\texttt{tr} > \ : \ ((\mathbf{TR}, F), w)} \tag{7.41}$$

$$\frac{\Gamma \vdash d : ((\mathbf{TR} \vee \circ, F), w)}{< \texttt{table} > d < /\texttt{table} > \ : \ ((\bullet, F), w)} \tag{7.42}$$

As the last core document element the server side call is treated. A call element may only contain actual parameter elements. This is ensured syntactically. The special sign $\varepsilon_{\texttt{act}}$ acts as an empty parameter list if necessary. It has the empty web signature as call type. Typing rule 7.45 makes it possible that several actual parameter elements uniquely provide the parameters for a server side call. Rule 7.43 specifies, that a server call can target an include server page only. The call element inherits the targeted include server page's document fragment type, because this page will replace the call element if it is called.

$$\frac{\Gamma \vdash l : w \rightarrow D \qquad \Gamma \vdash as : v \qquad \vdash v < w}{\Gamma \vdash < \texttt{call} \quad \texttt{callee} = "l" > as < /\texttt{call} > \ : \ D} \tag{7.43}$$

$$\frac{}{\Gamma \vdash \varepsilon_{\texttt{act}} : \emptyset} \tag{7.44}$$

$$\frac{\Gamma \vdash as : w \qquad \Gamma \vdash e : T \qquad l \notin (dom\ w)}{\Gamma \vdash \begin{array}{l} <\texttt{actualparam}\ \ \texttt{param} = "l" > \\ \qquad e \\ </\texttt{actualparam}> as\ :\ w \cup \{(l \mapsto T)\} \end{array}} \tag{7.45}$$

With the typing rule 7.46 and 7.49 arbitrary document fragment may become an include server page, thereby the document fragment's type becomes the server page's result type. A document fragment may become a complete web page by typing rules 7.47 and 7.49 if it has no specific layout type, i.e. is neutral or merely visible, and furthermore is not intended to be used inside forms. The resulting server page obtains the complete type as result type. Both include server page cores and web server page cores start with no formal parameters initially. With rule 7.48 parameters can be added to server page cores. The rule's premises ensure that a new formal parameter must have another name than all the other parameters and that the formal parameter is used in the core document type-correctly. A binding of a type to a new formal parameter's name is erased from the type environment.

$$\frac{\Gamma \vdash d : D \qquad d \in \textbf{dynamic}}{\Gamma \vdash <\texttt{include}> d </\texttt{include}> : \emptyset \to D} \tag{7.46}$$

$$\frac{\Gamma \vdash d : ((\bullet \vee \circ, \Updownarrow \vee \Uparrow), \emptyset) \qquad t \in \textbf{strings} \qquad d \in \textbf{dynamic}}{\Gamma \vdash \begin{array}{l} <\texttt{html}> \\ \qquad <\texttt{head}><\texttt{title}> t </\texttt{head}></\texttt{title}> \\ \qquad <\texttt{body}> d </\texttt{body}> \\ </\texttt{html}> : \emptyset \to \Box \end{array}} \tag{7.47}$$

$$\frac{\Gamma \vdash l : T \qquad \Gamma \vdash c : w \to D \qquad l \notin (dom\ w)}{\Gamma\backslash(l \mapsto T) \vdash \begin{array}{l} <\texttt{param}\ \ \texttt{name} = "l"\ \ \texttt{type} = "T"/> \\ c : (w \cup \{(l \mapsto T)\}) \to D \end{array}} \tag{7.48}$$

$$\frac{\Gamma \vdash l : P \qquad \Gamma \vdash c : P \qquad c \in \textbf{websig-core}}{\Gamma\backslash(l \mapsto P), \{(l \mapsto P)\} \vdash <\texttt{nsp}\ \ \texttt{name} = "l"> c </\texttt{nsp}>\ :\ \Diamond} \tag{7.49}$$

A server page core can become a well-typed server page by rule 7.49. The new server page name and the type bound to it are taken from the type environment and become the definition environment. An NSP system is a collection of NSP server pages. A single well-typed server page is already a system. Rule 7.50 specifies system compatibility. Rule 7.51 specifies system completeness. Two systems are compatible if they have no overlapping server page definitions. Furthermore the server pages that are defined in one system and used in the other must be able to process the data they receive from the other system, therefore the types of the server pages defined in the one system must be subtypes of the ones bound to their names in the other's system type environment.

$$\frac{\begin{array}{c} s_1, s_2 \in \textbf{system} \quad (dom\ \Delta_1) \cap (dom\ \Delta_2) = \emptyset \\ ((dom\ \Gamma_2) \lhd\ \Delta_1) < ((dom\ \Delta_1) \lhd\ \Gamma_2) \\ ((dom\ \Gamma_1) \lhd\ \Delta_2) < ((dom\ \Delta_2) \lhd\ \Gamma_1) \\ \Gamma_1, \Delta_1 \vdash s_1 : \diamond \qquad \Gamma_2, \Delta_2 \vdash s_2 : \diamond \end{array}}{((dom\ \Delta_2) \lhd \Gamma_1) \cup ((dom\ \Delta_1) \lhd \Gamma_2)\ ,\ \Delta_1 \cup \Delta_2\ \vdash s_1\ s_2\ :\ \diamond} \tag{7.50}$$

$$\frac{\begin{array}{c} \Gamma \in \mathbb{R} \\ (dom\ \Delta) \cap bound(s) = \emptyset \\ \Gamma, \Delta \vdash s : \diamond \end{array}}{\Gamma, \Delta \vdash s : \surd} \tag{7.51}$$

Typing rule 7.51 specifies when a well-typed system is complete. First, all of the used server pages must be defined, that is the type environment is a pure record type. Second server page definitions may not occur as bound variables somewhere in the system.

Theorem 7.6.1 *Core NSP type checking is decidable.*

Proof(7.6.1): Core NSP is explicitly typed. The Core NSP type system is algorithmic. Recursive subtyping is decidable. The least upper bound can be considered as a union operation during type checking - as a result a form content is considered to have a finite collection of types, which are checked each against a targeted server page if rule 7.37 is applied.□

Chapter 8

Conclusion

8.1 Related Work

8.1.1 Server Pages Technologies

Server side scripting technologies, like Active Server Pages (ASP) [110] [147], Python Server Pages (PSP) [4], and Java Server Pages (JSP) [143] allow to embed script code into HTML, or more generally XML. The Java Server Pages Standard Tag Library (JSTL) [51] improves JSP technology by providing tag support for typical actions. None of these technologies offer NSP's static type checking.

8.1.2 The "System Calls User" Approach

The language Mawl (Mother of All Web Languages) [7][8][106] is a server script approach with a sophisticated concept for form presentation. The definition language for forms is an HTML dialect. The scripting language is oriented towards the programming language C. Form presentation to the user is done in a "system calls user" way, i.e. a procedure is called in the server script. The actual parameters that are passed to the procedure are the data presented to the user, the return value of the procedure is the data entered by the user. This way Mawl allows for seamless integration of a script organizing session concept. The input type and output type of a form must be declared. Mawl supports static type checking of these types. Iterator tags, indexing for list data, and a dot notation for record data is available in the form description language for data presentation. Mawl separates between service logic code and markup language from the outset. Based on this device independency [23] is introduced to a certain extent: forms cannot be described in the aforementioned HTML dialect only [6], but in the Phone Markup Language[1] (PML) [151], too. In the "system calls user" approach the overall control flow is prescribed by the server

[1]PML allows for describing documents that are served over a telephone by special telecommunication portal middleware. PML is one of the precursors of Voice XML [118].

script code, because the return point of a form presentation is fixed. Form presentations are treated like procedure calls: after form processing the control flow returns to the call point. This abandons the core paradigm of hypertext, where a page may encompass several links and forms each targeting different locations. In order to add flexibility, Mawl supports multiple submit buttons. In the case that a form should be able to target different server actions, the receiving script can analyze, which submit button has been pressed, and can branch the control flow appropriately [44]. That is the purpose of usual HTML multiple submit buttons, too. However the resulting designs relying on such control flow branching must be considered flawed. These systems seriously suffer an "ask what kind" design antipattern resulting in high coupling and low cohesion. The situation is even worse, if multiple forms should target different server actions. These multiple forms must be emulated by one single superform and several submit capabilities. The output type of the superform must encompass the output types of all emulated forms, which has to be considered a further design flaw. We use the term "ask what kind" as a antipattern name in a generalized sense. Assume that a form containing several submit capabilities should be reused. Then every scripting code that reuses the form is responsible to branch the control flow correctly - high coupling. This is an error-prone pattern. Furthermore now consider a change to the submit buttons in the forms. Now it is necessary to be aware of all the distributed hard-wired case structures and to fix them accordingly - low cohesion. The term "Don't ask what kind" [36][37] has be coined in the object orientation community for usage of polymorphism as a basic pattern of assigning object responsibilities [107] for similar reasons.

The project Bigwig [18] declares itself as an intellectual descendant [15] of the Mawl project. Bigwig provides higher order templates [17][157]. However the motivation for higher order templates in Bigwig is different from the motivation of higher order server pages in NSP. In [18] gaining the flexibility of print-like statements in script-centered languages is given as a reason for the concept of higher order templates - in Bigwig no iterator tags are used in form descriptions as in Mawl. In contrast, in NSP higher order server pages are motivated very targeted as an enabling technology for maintainable solutions for typical design problems. Bigwig uses the non-standard notions of gaps and plugs for formal and actual parameters of form templates. A formal definition based on data flow analysis [131] is given for the type system of the language DynDoc, which is a sublanguage of Bigwig [17][157]. The defined static semantics guarantees user interface description safety. The Powerforms technology [16] is part of the Bigwig project. Powerforms defines a declarative language for expressing form input field formats and interdependencies between form input fields. Powerforms allows for the generation of client side code for ensuring the declared constraints. There is also a Java-based successor to Bigwig, called JWIG [31][32].

The rule-based CGI programming language Guide [109] supports a template mechanism that is similar to the one of Bigwig. However Guide does not support static type checking.

8.1.3 Web Application Frameworks

The NSP approach must not be mistaken to be a variant of architectural approaches. These web application frameworks target separation of presentation, content and logic to different extent, whereas they follow an approach to web application architecture that has become known as Model View Controller (MVC) architecture or Model 2 architecture. Further objectives of these web application frameworks can be integration with enterprise application frameworks, rapid development by providing an integrated development environment, support for internationalization, support for security, or support for dynamic form field validation. A prominent commercial MVC web application framework is the Sun ONE Application Framework (JATO) [167]. The most widespread application servers according to [149], namely Oracle9iAs, IBM WebSphere, and BEA WebLogic, come along with MVC web application frameworks, too. WebMacro [84] is an early open source project that allows the separation between Java and HTML. Other prominent open source web application frameworks are Struts [50] and Cocoon [39] - both hosted by the Jakarta project. Wafer (Web Application Research Project) [175] is a research project that collects, investigates and compares existing open source web application frameworks.

8.1.4 Web Services

Web Services [178] are widely regarded as a major technological shift in the usage of the web. Web services are primarily discussed in the B2B domain. Web services possess a type system, the Web service definition language [33]. At first glance, web services may seem inspired from and similar to user interfaces, using the same protocol, namely HTTP [14]. With respect to the NSP static type checking we can identify a clear difference between web services and HTTP/HTML user interfaces: in web services there is no necessary type relation between different messages, although there may be a type system. The types of different messages can be chosen freely according to the needs of the business case. In HTML dialogues however, if the user is supposed to send a form with data to the server, then a page containing this form must have been previously sent to the user. More generally speaking, we have a necessary relation between a typed user request and a systems response before that request. This relation is based in the mechanism itself.

In other words, in HTML dialogues, the type information is not transmitted once and used for all subsequent interactions, but it is transmitted before every typed request, and it is directly transformed into the displayed form, which the user actually fills out. This is not an accidental design mismatch between web services and HTML/HTTP dialogues, but a fundamental difference caused by the fact that web services are accessed automatically, and HTML dialogues are designed for end users.

8.1.5 XML Technologies

The XForms standard [69][68] defines the successor to HTML forms. XForms offers tag support for gathering structured data as XML documents. Data are transported as XML. In XForms data is separated from presentation from the outset, furthermore the XForms controls has to be considered platform independent, i.e. device independency is targeted. It is possible to specify constraints on data gathered by a form. Thereby the XForms approach does not only allow constraints on type correctness and required data entry, but allows to specify arbitrary validations and calculations with respect to data entered by the user. An event model defines hooks for handling violations of the specified constraints. In order to address parts of XML data in expressions XForms relies on XPath [35]. In HTML/XHMTL only plain unidirectional links are possible. Though a limited number of different link behaviors is available, important simple notions like e.g. a desired page decomposition are not defined for HTML/XHTML. XLink [52] overcomes these limitations in a general XML setting, whereas it has been influenced by [99]. XInclude [114] improves XLink with respect to document decomposition by defining a processing model for merging documents.

8.1.6 Functional Programming Language Approaches

The comprehensive WASH project offers the two technologies WASH/CGI [170] and WASH/HTML [168][169] for web authoring based on the functional programming language Haskell. WASH/CGI offers static checks for user interface type safety, WASH/HTML offers static checks for a weak user interface description safety. WASH/CGI is a domain specific language for server-side Web scripting. Dynamic generation of HTML output is monad-based. The form presentation concept encompasses a callback mechanism that allows for full design flexibility. WASH/CGI programs are dynamically type-safe: untrapped errors cannot occur, because necessary server-side reaction to user entered data that cannot be parsed is enforced. A compositional approach to specifying form elements is offered. WASH/HTML defines a Haskell combinator library for dynamic XML coding. For WASH/HTML systems full XML validity but only a limited form of HTML/XHTML validity can be guaranteed statically. In [120] a Haskell library for CGI scripting is proposed. There is also a server pages technology available for Haskell [122]. Haskell Server Pages guarantee well-formedness of XML documents. The small functional programming language XMλ [162] is designed to ensure full XML validity [121]. XMλ is based on XML documents as basic datatypes. The approach given by [120][122][162][121] comes along with a client side scripting technology for Haskell [119]. Yet another project in the context of Haskell is [180], which investigates two different approaches. In the first approach a library for XML processing arbitrary documents is provided. Thereby well-formedness of XML documents is ensured. The second is a type-based translation framework for XML documents with respect to a given DTD, which guarantees full XML validity.

The LAML (Lisp Abstract Markup Language) project [133][134][135] proposes web programming based on Scheme. Two other Scheme based web publishing systems are proposed in [148] and [81]. Beyond this it is shown in [81] how the features of a Sheme extension can be exploited for an efficient web server implementation. A conceptual basis for the continuation-style of functional web publishing technologies is given by [94] as a generalized notion of monads.

8.1.7 Logic Programming Language Approaches

In [86][87] a mature CGI programming library for the functional-logic multi-paradigm language Curry is proposed. A notion of submit button event handler is introduced and enables in effect a callback-style programming model. A specific abstract data type is used for references to input fields, which enables compile-time checks with resepct to input field naming. Pillow [25][24] is a CGI library for the CIAO Prolog system. LogicWeb [161] is a toolkit for even improved amalgamation of logic programming and web programming. Neither the Pillow approach nor the LogicWeb approach offer user interface type safety.

8.1.8 Web Application Reverse Engineering

The tool presented in [90][89][88] analyses source code and pages of a web application and generates an architecture diagram that visualizes the interactions between static pages, active ASP or JSP pages and other software components by arrows. The same support is offered by the tool WARE (Web Application Reverse Engineering) [112] for ASP and PHP based systems. A technique for recovering navigational structure and a conceptual model from a web application without tool support is described in [5]. An example for a tool that can track the change history of a web site is given in [22].

8.2 Further Work

Formal Parameter Requirement Specification

In the NSP approach it is possible to place requirements on the single formal parameters of web signatures. It is possible to require data entry by the user, a widget or a special kind of widget. The fulfillment of the requirements is statically ensured, thereby the NSP active controls are enabling technology. The possibility to place requirements on formal parameters can be generalized to arbitrary constraints. A suitable declarative constraint language can be defined, from which client-side checks are generated. The approach can be elaborated for array parameters.

Improved Design Recovery

The information extracted by JSPick from a Java Server Pages based system can be analyzed with respect to several classes of potential sources of error as already explained in 6.1. The amount of extracted information can be improved, i.e. lower and upper bounds for generated controls can be inferred.

Integration with Generator Type Safety

The notion of generator type safety is defined for a generic programming [47] mechanism that integrates parametric polymorphism and static introspection. The new generic programming mechanism has been proposed in [64]. The mechanism is powerful enough to provide simple and intuitive solutions to typical crosscutting problems, like e.g. transparent data access layers [9], as reusable components. Thereby the proposed balanced mechanism allows for static checks of generator type safety. The mechanism outlined above can be integrated in the NSP server pages concept. As an example, with the resulting technology it will be possible to implement a generic data form: given a class description a generic include server page generates a form with input capabilities for all attributes of the class.

8.3 Summary

The NSP project investigates client page description safety and client page type safety with respect to server pages technology. The contributions are

- parameterized server pages,
- higher order server pages,
- server-side calls to server pages,
- direct tag support for user defined programming language types in writing forms,
- virtual exchange of programmed objects across the web user agent,
- dynamically type safe direct input controls, statically ensured automatic constraint checks,
- introduction and formal definition of a tool for code-structure sensitive recovery of web signatures and form types from server pages based presentation layers,
- formal definition of static client page type and description correctness as a Per Marin-Löf style type system,
- and the integration of a scripting technology with a modeling technique.

Thereby the results are programming language independent.

Appendix A

Integration with Form-Oriented Analysis

In this appendix an example transformation of a system feature that is specified as a form-oriented analysis model into NSP technology is given. All model elements of form-oriented analysis that are used in this appendix are defined in the dissertation of Gerald Weber [178]. The system feature is modeled by a form chart with respect to a layered data model , i.e. a data dictionary and a semantic data model. A form chart can be transformed into server pages technology in several different ways, i.e. yielding several different system designs. There is one canonical transformation. This transformation maps each server action and each client page to one server page. The message types of server actions and client pages, which are specified in the data dictionary, become the web signatures. The server page of a server action hosts business logic implementation only and the server page of a client page has the only purpose to present data and create new data input capabilities. A server page of a server action prepares the data that has to be presented in the next step and branches the control flow with respect to the enabling conditions of the outgoing transitions. Forward engineering tools can be build that realize the canonical transformation by generating a complete executable prototype system for a given form chart and layered data model, whereas the prototype system provides protected regions in all kinds of generated server pages as well-defined hooks. The tool Gently [56][67] is such a forward engineering tool for generating Java Server Pages based systems. Again the Gently approach is presented in the dissertation of Gerald Weber [178].

The canonical transformation guarantees maximum reusability and low coupling and allows for an immediate exploitation of further form-oriented dialogue elements like e.g. dialogue constraints. However it yields not the most succinct kind of design. Design-decisions may lead to other justifiable transformations, for example to such transformations that foster high cohesion. Actually for example in this appendix a very straightforward transformation to NSP technology has been chosen, that always implements a server action and an adjacent client page as one server page.

The NSP concept of functional decomposition of server pages can be exploited to implement a server action's branching to client pages. The NSP concept of higher order server pages can be exploited to solve some typical design problems that often arise in web application architecture but can be identified early in form charts as has been explained in section 4.3 with listing 4.2 and Figure 4.4. However the aim of this chapter is just to give some confidence that the NSP technology is tightly integrated with the overall form-oriented approach by providing a somewhat larger example from a real system that is currently under development. The chosen transformation is already sufficient for this purpose. It can be seen how web signatures of a system dialogue become NSP web signatures.

The appendix proceeds as follows. First the vision of a software system is given. Then a single desired feature in this system is described informally and then specified using form-oriented analysis. In section A.2 the mapping of this specification to NSP server pages is given.

A.1 Problem Description

A.1.1 Vision

It is the vision to develop a combined CSCW/project management tool for the software engineering process EASE.

EASE (Education for Actual Software Engineering) [65] is a software engineering process model for higher education. It is the first such process that is designed from scratch, whereas it is oriented towards proven concepts from mature andragogical methodologies like Collaborative Learning [21], Action Learning [154], and Entraînement Mental [30].

The project management tool PEASE (Platform for EASE) is currently under development [93]. The project planning phase, comprising competitive product analysis [177] as a basis for a positive stop-or-go decision, has already been finished [92]. The PEASE platform is a web-enabled project management tool that is oriented towards groupware [82]. There are already a couple of such systems, the most prominent one is probably [136] which arose in the open source community [152]. A detailed explanation of EASE is given in [65]. PEASE supports all activities found in the EASE process architecture, especially the micro process of EASE, which can be loosely compared to the iteration planning of the Extreme Programming [11] software development approach. The EASE micro process is a fast iterative alternation of meetings and plannings. In every meeting tasks are found, sorted, discussed, and assigned to teams. Every week new teams are formed; small changing groups foster the communication between students. Thereby the project's progress is tracked. In this appendix a single feature of the future PEASE platform for managing tasks serves as example. The task manager feature allows removing tasks and deleting or adding team members. It is most probably used after the assignment of tasks , in order to adjust some recently entered data.

A.1.2 Task Manager Feature Description

The task manager feature allows removing tasks, deleting team members from tasks, and adding new team members to tasks. At the entry page of the task manager feature the user is shown a single select menu of all current tasks. She can select a task and choose between two options, namely removing the task and editing the task[1]. For this purpose two submit buttons are offered. Pressing a submit button will be successful only if the user has selected at least one task. If the user has chosen to remove a task from the task list first a message page is presented to the user. The user is asked to acknowledge or otherwise to redeem that she actually wants to remove the chosen task. Based on the user's decision the task is removed and the dialogue returns to the task manager entry page.

If the user chooses to edit a task from the entry page's task list a page is shown that contains a link and a form. With the link it is possible to return to the entry page without changing anything. The form contains two multiple select menus. In the first menu the team members of the selected task are listed. Selected team members will be removed from the task on submit. In the second menu all project members that are not assigned to the task are listed. Selected project members will be added to the task as new team members on submit. On submit the necessary update operations are executed. Then a message page is presented. It contains the name of the current task and the updated list of team members and it invites the user to control the result. The user has the option to return to the entry page or to edit the task again.

A.1.3 Task Manager Feature Specification

The task manager feature that has been described in section A.1.2 is modeled by the form chart given in Figure A.1 and a layered data model given in Figure A.2. The persistent data is given by the semantic data model: every task has a name and a list of team members. The semantics of the data dictionary types, i.e. the message types, and their interplay with semantic data types is explained in [178]: every type introduced in the semantic data model is available in the data dictionary as opaque reference type, which consists of the keys to the persistent data objects.

We do not give a complete dialogue constraint specification. We pick the dialogue constraints for one page/server transition as an example. A page/server transition is annotated with an enabling condition, a client output constraint, and a server input constraint.

[1]The task manager feature is a feature in the sense of feature orientation [60]. That is the specification of the pages that make up the respective dialogue are partial specifications. In a complete system specification the single pages may offer more information and more interaction capabilities. For example the page just described will have a link for the registration of a new task in the complete system specification.

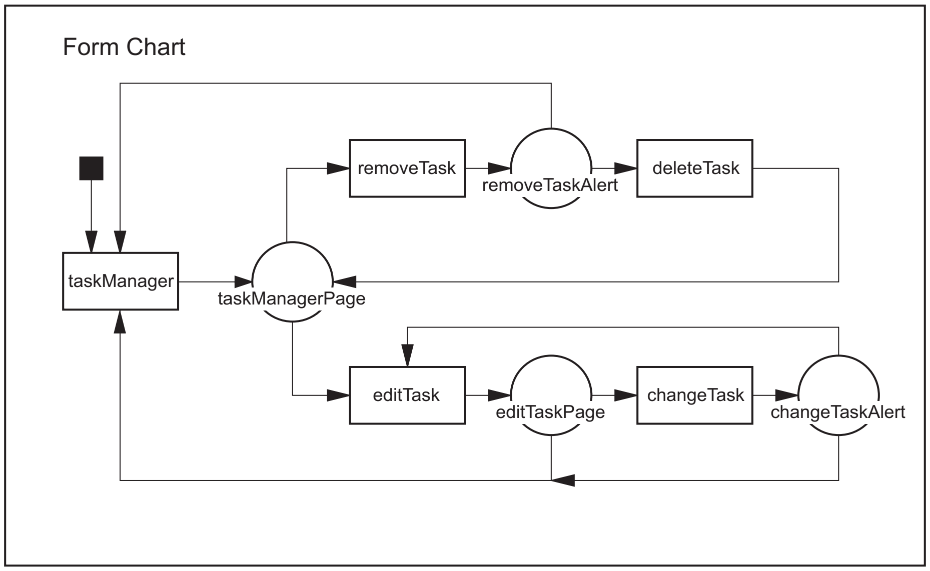


Figure A.1: Form Chart Diagram. The figure visualizes a task manager feature of a combined CSCW/project management tool.

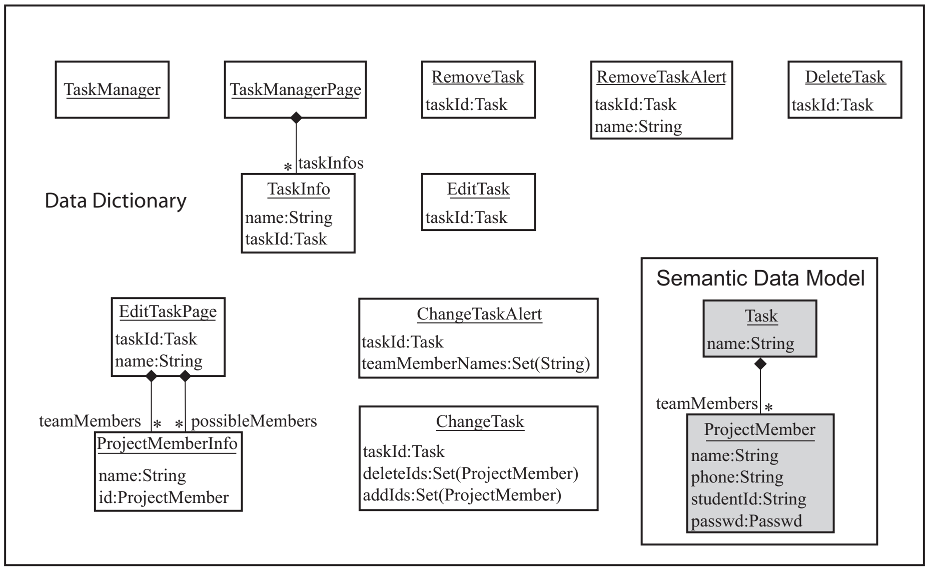


Figure A.2: Layered Data Model. The figure visualizes the form-oriented data model which underlies a task manager feature of a combined CSCW/project management tool. The data model consists of two layers, a data dictionary and a semantic data model. Data model types are used as opaque reference types in the data dictionary.

Listing A.1

```
editTaskPage to changeTask{
  enabling:
  clientOutput:
    changeTask.taskId=editTaskPage.taskId
    editTaskPage.teamMembers->includesAll(changeTask.deleteIds)
    editTaskPage.possibleMembers->includesAll(changeTask.addIds)
  serverInput:
    ChangeTask.deleteIds->notEmpty or ChangeTask.addIds->notEmpty
}
```

We consider the page editTaskPage in Figure A.1, which offers the user a form with two select menus for removing and adding team members. We consider the transition that is triggered by submitting the form, i.e. the transition to the changeTask server action. In listing A.1 the constraints for this transition are given. The enabling condition is empty. We consider the constraints of the client output specification next. First the current taskId must be passed unchanged to the server action. Then the team members that have been chosen to be deleted must actually belong to the list of team members presented to the user. The team members that should be added to the team are constrained analogously. The server input constraint states that the server action should only be executed if the user has selected at least one change.

A.2 Mapping A Form-Oriented Specification to NSP

The form-oriented feature specification given in section A.1.3 is implemented by the NSP server pages given in listings A.2 to A.5. Under the chosen transformation always a server action and the adjacent client page is implemented by one server page. The server page DeleteTask implements the effect of data deletion, but it does neither implement the retrieval of information for the client page taskManagerPage nor its presentation. For this purpose it reuses the implementation provided by the server page TaskManager by forwarding to it. The transformation of server actions and client pages is summarized in the informal equation A.1.

$$\begin{array}{ccl}
\texttt{taskManager taskManagerPage} & \mapsto & \texttt{TaskManager} \\
\texttt{removeTask removeTaskAlert} & \mapsto & \texttt{RemoveTask} \\
\texttt{deleteTaskManager taskManagerPage} & \mapsto & \texttt{DeleteTask} \rightsquigarrow \texttt{TaskManager} \\
\texttt{editTask editTaskPage} & \mapsto & \texttt{EditTask} \\
\texttt{changeTask changeTaskAlert} & \mapsto & \texttt{ChangeTask}
\end{array} \tag{A.1}$$

The message types of server actions are directly mapped to web signatures of NSP server pages.

Some message type parameters of client pages are mapped to formal parameters of a server page. Others are mapped to local variables of a server page or Java expressions. The mapping is pictured by the informal equation A.2[2].

$$\begin{array}{l}
\texttt{TaskManagerPage.taskInfos.taskId} \mapsto \texttt{TaskManager.taskIds} \\
\texttt{TaskManagerPage.taskInfos}[\texttt{i}]\texttt{.name} \mapsto \texttt{.getTaskName}(\texttt{TaskManager.taskIds}[\texttt{i}]) \\
\texttt{RemoveTaskAlert.taskId} \mapsto \texttt{RemoveTask.}<\texttt{param}>\texttt{taskId} \\
\texttt{RemoveTaskAlert.name} \mapsto \texttt{.getTaskName}(\texttt{RemoveTask.}<\texttt{param}>\texttt{taskId}) \\
\texttt{EditTaskPage.taskId} \mapsto \texttt{EditTask.}<\texttt{param}>\texttt{taskId} \\
\texttt{EditTaskPage.name} \mapsto \texttt{.getTaskName}(\texttt{EditTask.}<\texttt{param}>\texttt{taskId}) \\
\texttt{EditTaskPage.teamMembers.id} \mapsto \texttt{EditTask.teamMemberIds} \\
\quad \texttt{EditTaskPage.teamMembers.name}[\texttt{i}] \\
\quad \mapsto \texttt{getProjectMemberName}(\texttt{EditTask.teamMemberIds}[\texttt{i}]) \\
\texttt{EditTaskPage.possibleMembers.id} \mapsto \texttt{EditTask.possibleMemberIds} \\
\quad \texttt{EditTaskPage.possibleMembers.name}[\texttt{i}] \\
\quad \mapsto \texttt{getProjectMemberName}(\texttt{EditTask.possibleMemberIds}[\texttt{i}]) \\
\texttt{ChangeTaskAlert.taskId} \mapsto \texttt{ChangeTask.}<\texttt{param}>\texttt{taskId} \\
\texttt{ChangeTaskAlert.teamMemberNames}[\texttt{i}] \mapsto \texttt{ChangeTask.teamMemberNames}
\end{array} \tag{A.2}$$

Note how the NSP submit button concept is used for realizing several possible page/server transitions of a client page directly. For example the submit buttons in listing A.2 target different server pages.

The output constraints for project members of listing A.1 are easily fulfilled, because only select menus has been chosen to implement the lists of team members and possibly project members: the user cannot choose values that lead to a violation of the constraint. The server input constraint of listing A.1 is ignored in the current implementation. It has status TBD [95] per definition, i.e. only in a complete specification will be decided whether the server input constraint is ensured by client side scripting or additional dialogue.

[2]Formal server page parameters are marked by a juxtaposed <`param`>-tag in equation A.2. The methods in equation A.2 are class methods of the imported class CustomerBase. The class name CustomerBase is dropped. The semantics of the methods is implicitly specified by equation A.2, too.

Listing A.2

```
<nsp name="TaskManager">
  <html>
    <head><title>Task Manager</title></head><java>
    import myBusinessModel.CustomerBase;</java>
    <body><java>
      TaskId taskIds[]=CustomerBase.getTaskIds();</java>
      <form callee="editTask">
        <select param="taskId"><java>
          for (i=0;i<taskIds.length;i++){</java>
            <option>
              <value>
                taskIds[i]
              </value>
              <label>
                CustomerBase.getTaskName(taskIds[i])
              </label>
            </option><java>
          }</java>
        </select><br>
        <submit callee="removeTask"><label>Remove</label></submit><br>
        <submit callee="editTask"><label>Edit</label></submit><br>
      </form>
    </body>
  </html>
</nsp>
```

Listing A.3

```
<nsp name="RemoveTask">
  <param type="TaskId" name="taskId"></param>
  <html><head><title>Remove Task Alert</title></head>
  <body>
    Do you really want to remove task &nbsp;
    <javaexpr>CustomerBase.getTaskName(taskId)</javaexpr>
    &nbsp; ? <br>
    <form callee="deleteTask">
      <submit callee="deleteTask">
        <label>YES</label>
        <hidden>taskId</hidden>
      </submit>
      <submit callee="taskManager">
        <label>NO</label>
      </submit>
    </form>
  </body>
</nsp>
```

Listing A.4

```
<nsp name="EditTask">
  <param type="TaskId" name="taskId"></param>
  <html><head><title>Edit Task</title></head>
  <body>
    You want to edit task &nbsp;
    <javaexpr>CustomerBase.getTaskName(taskId)<javaexpr>
    &nbsp; .<br>
    <link callee="taskManager">Back to task manager.</link><br>
    <form callee="changeTask"><java>
      ProjectMemberId[] teamMemberIds = CustomerBase.getTeamMemberIds(taskId);
      ProjectMemberId[] possibleMemberIds =
        CustomerBase.getProjectMemberIdsUnequalTo(teamMemberIds);
      if (teamMemberIds.length>0) {</java>
        Remove the following team members from the task:<java>
        <select multiple param="deleteIds"><java>
          for (i=0;i<teamMemberIds.length;i++) {</java>
          <option>
            <value>
              teamMemberIds[i]
            </value>
            <label>
              <javaexpr>
              CustomerBase.getProjectMemberName(teamMemberIds[i])
              </javaexpr>
            </label>
          </option>
        </select>
      }</java>
      if (possibleMemberIds.length>0) {</java>
       Add the following team members to the task:<java>
        <select multiple param="addIds"><java>
          for (i=0;i<possibleMemberIds.length;i++) {</java>
          <option>
            <value>
              possibleMemberIds[i]
            </value>
            <label>
              <javaexpr>
              CustomerBase.getProjectMemberName(possibleMemberIds[i])
              </javaexpr>
            </label>
          </option>
        </select>
      }</java>
      <hidden param="taskId">taskId</hidden>
      <submit></submit>
    </form>
  </body>
</nsp>
```

Listing A.5

```
<nsp name="ChangeTask">
  <param type="TaskId" name="taskId"></param>
  <param type="ProjectMemberId[]" name="deleteIds"></param>
  <param type="ProjectMemberId[]" name="addIds"></param>
  <html><head><title></title></head>
  <body><java>
    CustomerBase.deleteTeamMembers(deleteIds);
    CustomerBase.addTeamMembers(addIds);
    </java>
    You changed the task &nbsp;
    <javaexpr>CustomerBase.getTaskName(taskId)</javaexpr>
    &nbsp;.<br>
    Please check if the changes are correct !<br>
    The task are assigned the following team members:<br><java>
    String[] teamMemberNames = CustomerBase.getTeamMemberNames(taskId);
    for (i=1;i<teamMemberIds.length;i++){</java>
      <javaexpr>i</javaexpr>.
      <javaexpr>teamMemberNames[i]</javaexpr>
      <br><java>
    }</java>
    <form callee="taskManager">
      <hidden param="taskId">taskId</hidden>
      <submit callee="taskManager"><label>OK</label></submit>
      <submit callee="editTask"><label>Edit Task Again</label></submit>
    </form>
  </body>
</nsp>
```

Listing A.6

```
<nsp name="DeleteTask">
  <param type="TaskId" name="taskId"></param>
  <html><head><title></title></head>
  <body><java>
    CustomerBase.deleteTask(taskId);</java>
    <forward callee="taskManager"></forward>
  </body>
</nsp>
```

Appendix B

NSP Language Definition

B.1 Context Free Grammar

`string`	`::=`	s $\in$ **String**
`id`	`::=`	l $\in$ **Label**
`parameter-type`	`::=`	t $\in \mathbb{T} \cup \mathbb{P}$
`supported-type`	`::=`	t $\in \mathbb{B}_{supported}$
`system`	`::=`	`page | system system`
`page`	`::=`	`<nsp name="id"> websig-core </nsp>`
`websig-core`	`::=`	`param websig-core | webcall | include`
`param`	`::=`	`<param name="id" type="parameter-type"/>`
`webcall`	`::=`	`<html> head body </html>`
`head`	`::=`	`<head><title> strings </title></head>`
`strings`	`::=`	ε `| string strings`
`body`	`::=`	`<body> dynamic </body>`
`include`	`::=`	`<include> dynamic </include>`

```
     dynamic  ::=    dynamic dynamic
                |    ε | string
                |    ul | li
                |    table | tr | td
                |    call
                |    form | object | hidden | submit
                |    input | checkbox
                |    select | option
                |    expression
                |    code

          ul  ::=    <ul> dynamic </ul>
          li  ::=    <li> dynamic </li>

       table  ::=    <table> dynamic </table>
          tr  ::=    <tr> dynamic </tr>
          td  ::=    <td> dynamic </td>

        call  ::=    <call callee="id"> actualparams </call>

actualparams  ::=    ε_act | actualparam actualparams

 actualparam  ::=    <actualparam param="id"> expr </actualparam>

        form  ::=    <form callee="id"> dynamic </form>

      object  ::=    <object param="id"> dynamic </object>

      hidden  ::=    <hidden param="id"> expr </hidden>

      submit  ::=    <submit/>

       input  ::=    <input type="supported-type" param="id"/>

    checkbox  ::=    <checkbox param="id"/>
    select   ::=     <select param="id"> dynamic </select>

    option   ::=     <option>
                         <value> expr </value>
                         <label> expr </label>
                     </option>

expression   ::=     <expression> expr </expression>

      code   ::=     <code> com </code>

       com   ::=     </code> dynamic <code>
                  |  com ; com
                  |  if expr then com else com
                  |  while expr do com
                  |  stat

      stat   ::=     id := expr

      expr   ::=     id  | expr.id | expr[expr]
```

B.2 Types

B.2.1 Programming Language Types

$$\mathbb{T} \;=\; \mathbb{B} \cup \mathbb{V} \cup \mathbb{A} \cup \mathbb{R} \cup \mathbb{Y}$$

$$\begin{array}{rll}
(\textit{basic types}) & \mathbb{B} & = \mathbb{B}_{primitive} \cup \mathbb{B}_{supported} \\
(\textit{primitive basic types}) & \mathbb{B}_{primitive} & = \{\texttt{int}, \texttt{float}, \texttt{boolean}\} \\
(\textit{supported basic types}) & \mathbb{B}_{supported} & = \{\texttt{int}, \texttt{Integer}, \texttt{String}\}
\end{array}$$

$$\begin{array}{rlll}
(\textit{type variables}) & \mathbb{V} & = & \{X, Y, Z, \ldots\} \\
 & & \cup & \{\texttt{Person}, \texttt{Customer}, \texttt{Article}, \ldots\} \\
(\textit{array types}) & \mathbb{A} & = & \{\, \texttt{array of } T \mid T \in \mathbb{T} \setminus \mathbb{A}\} \\
(\textit{record types}) & \mathbb{R} & = & \mathbf{Label} ⇻ \mathbb{T} \\
(\textit{recursive types}) & \mathbb{Y} & = & \{\, \mu\, X \,.\, R \mid X \in \mathbf{V} \,,\; R \in \mathbb{R} \,\}
\end{array}$$

B.2.2 Server Page Types

$$\begin{array}{rll}
(\textit{page types}) & \mathbb{P} & = \{\, w \rightarrow r \mid w \in \mathbb{W} \,,\; r \in \mathbb{C} \cup \mathbb{D} \,\} \\
(\textit{web signatures}) & \mathbb{W} & = \mathbf{Label} ⇻ (\mathbb{T} \cup \mathbb{P}) \\
(\textit{complete web page}) & \mathbb{C} & = \{\Box\} \\
(\textit{document fragment types}) & \mathbb{D} & = \mathbb{L} \times \mathbb{W} \\
(\textit{layout types}) & \mathbb{L} & = \mathbb{E} \times \mathbb{F} \\
(\textit{element types}) & \mathbb{E} & = \{\, \circ, \bullet, \mathbf{TR}, \mathbf{TD}, \mathbf{LI}, \mathbf{OP}\} \\
(\textit{form occurences}) & \mathbb{F} & = \{\, \Downarrow, \Uparrow, \Updownarrow \,\} \\
(\textit{system types}) & \mathbb{S} & = \{\, \Diamond, \surd \,\}
\end{array}$$

B.3 Subtyping Relation

B.3.1 Establishing Subtyping Rules

$$(reflexivity) \qquad \frac{}{\vdash T < T}$$

$$(array\ types) \qquad \frac{}{\vdash T < \texttt{array of}\ T}$$

$$\begin{pmatrix} record\ types \\ web\ signatures \end{pmatrix} \qquad \frac{T_j \notin \mathbb{B}_{primitive} \cup \mathbb{P} \quad j \in 1\ldots n}{\vdash \{l_i \mapsto T_i\}^{i \in 1\ldots j-1, j+1 \ldots n} < \{l_i \mapsto T_i\}^{i \in 1 \ldots n}}$$

$$(html\ types) \qquad \frac{T \in \mathbb{E}}{\vdash \circ\ <\ T}$$

$$(form\ occurences) \qquad \frac{T \in \mathbb{F}}{\vdash \Updownarrow <\ T}$$

B.3.2 Preserving Subtyping Rules

$$(array\ types) \qquad \frac{\vdash S < T}{\vdash \texttt{array of}\ S < \texttt{array of}\ T}$$

$$\begin{pmatrix} record\ types \\ web\ signatures \end{pmatrix} \qquad \frac{\vdash S_1 < T_1 \ldots \vdash S_n < T_n}{\vdash \{l_i \mapsto S_i\}^{i \in 1 \ldots n} < \{l_i \mapsto T_i\}^{i \in 1 \ldots n}}$$

$$(recursive\ types) \qquad \frac{\vdash S[{}^{\mu X.S}/_X] < T}{\vdash \mu X.S < T} \qquad \frac{\vdash S < T[{}^{\mu X.T}/_X]}{\vdash S < \mu X.T}$$

$$(page\ types) \qquad \frac{\vdash w' < w \qquad \vdash R < R'}{\vdash w \to R < w' \to R'}$$

$$\begin{pmatrix} dynamic \\ layout\ types \end{pmatrix} \qquad \frac{\vdash S_1 < T_1 \quad \vdash S_2 < T_2}{\vdash (S_1, S_2) < (T_1, T_2)}$$

B.4 Type Operators

$\_* \;:\; \mathbb{T} \rightarrow \mathbb{T}$

$$T* \equiv_{\mathtt{DEF}} \begin{cases} \texttt{array of}\ T & , T \notin \mathbb{A} \\ T & , else \end{cases}$$

$\_ \odot \_ \;:\; \mathbb{W} \rightarrowtriangle \mathbb{W}$

$w_1 \odot w_2 \equiv_{\mathtt{DEF}}$

$$\begin{cases} \perp & , \boldsymbol{if}\ \exists (l_1 \mapsto T_1) \in w_1 \bullet \exists (l_2 \mapsto T_2) \in w_2 \bullet\ l_1 = l_2 \wedge P_1 \in \mathbb{P} \wedge P_2 \in \mathbb{P} \\ \perp & , \boldsymbol{if}\ \exists (l_1 \mapsto T_1) \in w_1 \bullet \exists (l_2 \mapsto T_2) \in w_2 \bullet\ l_1 = l_2 \wedge T_1 \sqcup T_2\ undefined \\ & (\texttt{dom}\ w_2) \lhd w_1 \ \cup\ (\texttt{dom}\ w_1) \lhd w_2 \\ \cup & \left\{ \big(l \mapsto (T_1 \sqcup T_2)*\big) \mid (l \mapsto T_1) \in w_1 \wedge (l \mapsto T_2) \in w_2 \right\} \quad , \boldsymbol{else} \end{cases}$$

B.5 Environments and Judgements

$$\begin{array}{lll} \Gamma : \textbf{Label} \twoheadrightarrow (\mathbb{T} \cup \mathbb{P}) & = \mathbb{W} & (identifiers) \\ \Delta : \textbf{Label} \twoheadrightarrow \mathbb{P} & \subset \mathbb{W} & (page\ definitions) \end{array}$$

$$\begin{array}{ll} \Gamma \vdash e : \mathbb{T} \cup \mathbb{P} & e \in \textbf{expr} \\ \Gamma \vdash n : \mathbb{D} & n \in \textbf{com} \cup \textbf{dynamic} \\ \Gamma \vdash c : \mathbb{P} & c \in \textbf{websig-core} \\ \Gamma \vdash a : \mathbb{W} & a \in \textbf{actualparams} \\ \Gamma, \Delta \vdash s : \mathbb{S} & s \in \textbf{system} \end{array}$$

B.6 Typing Rules

$$\frac{\begin{array}{c}\Gamma \in \mathbb{R}\\ (dom\ \Delta) \cap bound(s) = \emptyset\\ \Gamma, \Delta \vdash s : \diamond\end{array}}{\Gamma, \Delta \vdash s : \surd} \tag{B.1}$$

$$\frac{\begin{array}{c}s_1, s_2 \in \textbf{system} \quad (dom\ \Delta_1) \cap (dom\ \Delta_2) = \emptyset\\ ((dom\ \Gamma_2) \lhd\ \Delta_1) < ((dom\ \Delta_1) \lhd\ \Gamma_2)\\ ((dom\ \Gamma_1) \lhd\ \Delta_2) < ((dom\ \Delta_2) \lhd\ \Gamma_1)\\ \Gamma_1, \Delta_1 \vdash s_1 : \diamond \qquad \Gamma_2, \Delta_2 \vdash s_2 : \diamond\end{array}}{((dom\ \Delta_2) \lhd \Gamma_1) \cup ((dom\ \Delta_1) \lhd \Gamma_2)\ ,\ \Delta_1 \cup \Delta_2\ \vdash s_1\ s_2\ :\ \diamond} \tag{B.2}$$

$$\frac{\Gamma \vdash l : P \qquad \Gamma \vdash c : P \qquad c \in \textbf{websig-core}}{\Gamma \backslash (l \mapsto P), \{(l \mapsto P)\}\ \vdash <\texttt{nsp}\quad \texttt{name} = "l" > c < /\texttt{nsp} >\ :\ \diamond} \tag{B.3}$$

$$\frac{\Gamma \vdash l : T \qquad \Gamma \vdash c : w \to D \qquad l \notin (dom\ w)}{\Gamma \backslash (l \mapsto T) \vdash \begin{array}{l} <\texttt{param}\quad \texttt{name} = "l" \quad \texttt{type} = "T"/> \\ c : (w \cup \{(l \mapsto T)\}) \to D\end{array}} \tag{B.4}$$

$$\frac{\Gamma \vdash d : ((\bullet \vee \circ, \Updownarrow \vee \Uparrow), \emptyset) \qquad t \in \textbf{strings} \qquad d \in \textbf{dynamic}}{\Gamma \vdash \begin{array}{l} <\texttt{html}> \\ \quad <\texttt{head}><\texttt{title}> t </\texttt{head}></\texttt{title}> \\ \quad <\texttt{body}> d </\texttt{body}> \\ </\texttt{html}> : \emptyset \to \Box \end{array}} \tag{B.5}$$

$$\frac{\Gamma \vdash d : D \qquad d \in \textbf{dynamic}}{\Gamma \vdash <\texttt{include}> d </\texttt{include}> : \emptyset \to D} \tag{B.6}$$

$$\frac{d \in \textbf{string}}{\Gamma \vdash d : ((\bullet, \Updownarrow), \emptyset)} \tag{B.7}$$

$$\frac{}{\Gamma \vdash \boldsymbol{\varepsilon} : ((\circ, \Updownarrow), \emptyset)} \tag{B.8}$$

$$\frac{\begin{array}{c} d_1, d_2 \in \textbf{dynamic}\\ \Gamma \vdash d_1 : (L_1, w_1) \quad \Gamma \vdash d_2 : (L_2, w_2) \quad L_1 \sqcup L_2 \downarrow \quad w_1 \odot w_2 \downarrow\end{array}}{\Gamma \vdash d_1\ d_2 : (L_1 \sqcup L_2, w_1 \odot w_2)} \tag{B.9}$$

$$\frac{\Gamma \vdash d : ((\mathbf{LI} \vee \circ, F), w)}{< \texttt{ul} > d < /\texttt{ul} > \ : \ ((\bullet, F), w)} \tag{B.10}$$

$$\frac{\Gamma \vdash d : ((\bullet \vee \circ, F), w)}{< \texttt{li} > d < /\texttt{li} > \ : \ ((\mathbf{LI}, F), w)} \tag{B.11}$$

$$\frac{\Gamma \vdash d : ((\mathbf{TR} \vee \circ, F), w)}{< \texttt{table} > d < /\texttt{table} > \ : \ ((\bullet, F), w)} \tag{B.12}$$

$$\frac{\Gamma \vdash d : ((\mathbf{TD} \vee \circ, F), w)}{< \texttt{tr} > d < /\texttt{tr} > \ : \ ((\mathbf{TR}, F), w)} \tag{B.13}$$

$$\frac{\Gamma \vdash d : ((\bullet \vee \circ, F), w)}{< \texttt{td} > d < /\texttt{td} > \ : \ ((\mathbf{TD}, F), w)} \tag{B.14}$$

$$\frac{\Gamma \vdash l : w \rightarrow D \qquad \Gamma \vdash as : v \qquad \vdash v < w}{\Gamma \vdash < \texttt{call} \quad \texttt{callee} = "l" > as < /\texttt{call} > \ : \ D} \tag{B.15}$$

$$\frac{}{\Gamma \vdash \varepsilon_{\texttt{act}} : \emptyset} \tag{B.16}$$

$$\frac{\Gamma \vdash as : w \qquad \Gamma \vdash e : T \qquad l \notin (dom\ w)}{\Gamma \vdash \begin{array}{l} < \texttt{actualparam} \quad \texttt{param} = "l" > \\ \qquad e \\ < /\texttt{actualparam} > as \ : \ w \cup \{(l \mapsto T)\} \end{array}} \tag{B.17}$$

$$\frac{\Gamma \vdash l : w \rightarrow \Box \qquad \Gamma \vdash d : ((e, \Downarrow), v) \qquad \vdash v < w}{\Gamma \vdash < \texttt{form} \quad \texttt{callee} = "l" > d < /\texttt{form} > : ((e, \Uparrow), \emptyset)} \tag{B.18}$$

$$\frac{\Gamma \vdash d : (L, w)}{\Gamma \vdash < \texttt{object}\ \texttt{param} = "l" > d < /\texttt{object} > \ : \ (L, \{(l \mapsto w)\})} \tag{B.19}$$

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash < \texttt{hidden}\ \texttt{param} = "l" > e < /\texttt{hidden} > \ : \ ((\circ, \Downarrow), \{(l \mapsto T)\})} \tag{B.20}$$

$$\frac{}{\Gamma \vdash < \texttt{submit}/ > \ : \ ((\bullet, \Downarrow), \emptyset)} \tag{B.21}$$

$$\frac{T \in \mathbb{B}_{supported}}{\Gamma \vdash \ <\texttt{input type} = \text{"}T\text{"} \ \texttt{param} = \text{"}l\text{"}/> : ((\bullet, \Downarrow), \{(l \mapsto T)\})} \tag{B.22}$$

$$\frac{}{\Gamma \vdash <\texttt{checkbox} \quad \texttt{param} = \text{"}l\text{"}/> \ : \ ((\bullet, \Updownarrow), \{(l \mapsto boolean)\})} \tag{B.23}$$

$$\frac{\Gamma \vdash d : \big((\mathbf{OP}, \Updownarrow), \{(l \mapsto \texttt{array of } T)\}\big)}{\Gamma \vdash \begin{array}{l} <\texttt{select} \quad \texttt{param} = \text{"}l\text{"}> \\ \qquad d \\ </\texttt{select}> \ : \ ((\bullet, \Downarrow), \{(l \mapsto \texttt{array of } T)\}) \end{array}} \tag{B.24}$$

$$\frac{\Gamma \vdash v \ : \ T \qquad \Gamma \vdash e : S \quad S \in \mathbb{B} \qquad l \in \textbf{Label}}{\Gamma \vdash \begin{array}{l} <\texttt{option}> \\ \qquad <\texttt{value}> v </\texttt{value}> \\ \qquad <\texttt{label}> e </\texttt{label}> \\ </\texttt{option}> : ((\mathbf{OP}, \Updownarrow), \{(l \mapsto \texttt{array of } T)\}) \end{array}} \tag{B.25}$$

$$\frac{\Gamma \vdash e : T \quad T \in \mathbb{B}}{\Gamma \vdash <\texttt{expression}> e </\texttt{expression}> \ : \ ((\bullet, \Updownarrow), \emptyset)} \tag{B.26}$$

$$\frac{\Gamma \vdash c : D}{\Gamma \vdash <\texttt{code}> c </\texttt{code}> \ : \ D} \tag{B.27}$$

$$\frac{\Gamma \vdash d : D}{\Gamma \vdash </\texttt{code}> d <\texttt{code}> \ : \ D} \tag{B.28}$$

$$\frac{\Gamma \vdash c_1 : (L_1, w_1) \qquad \Gamma \vdash c_2 : (L_2, w_2) \qquad L_1 \sqcup L_2 \downarrow \qquad w_1 \odot w_2 \downarrow}{\Gamma \vdash c_1 ; c_2 \ : \ (L_1 \sqcup L_2, w_1 \odot w_2)} \tag{B.29}$$

$$\frac{\Gamma \vdash e : \texttt{boolean} \quad \Gamma \vdash c_1 : D_1 \quad \Gamma \vdash c_2 : D_2 \qquad D_1 \sqcup D_2 \downarrow}{\Gamma \vdash \texttt{if } e \texttt{ then } c_1 \texttt{else } c_2 \ : \ D_1 \sqcup D_2} \tag{B.30}$$

$$\frac{\Gamma \vdash e : \texttt{boolean} \qquad \Gamma \vdash c : (L, w)}{\Gamma \vdash \texttt{while } e \texttt{ do } c \ : \ (L, w \odot w)} \tag{B.31}$$

$$\frac{\Gamma \vdash x : T \qquad \Gamma \vdash e : T \qquad T \in \mathbb{T}}{\Gamma \vdash x := e \ : \ ((\circ, \Updownarrow), \emptyset)} \tag{B.32}$$

$$\frac{(v \mapsto T) \in \Gamma}{\Gamma \vdash v : T} \tag{B.33}$$

$$\frac{\Gamma \vdash e : \{l_i \mapsto T_i\}^{i \in 1 \ldots n} \qquad j \in 1 \ldots n}{\Gamma \vdash e.l_j : T_j} \tag{B.34}$$

$$\frac{\Gamma \vdash e : \ \texttt{array of}\ T \qquad \Gamma \vdash i : \texttt{int}}{\Gamma \vdash e[i] : T} \tag{B.35}$$

B.7 Example Type Derivation I

Listing B.1

```
<nsp name="A">
   <param name="x" type="array of int"/>
   <include> ε </include>
</nsp>

<nsp name="B">
   <param name="x" type="int"/>
   <include>
      <li>
         <expression> x </expression>
      </li>
   </include>
</nsp>

<nsp name="C">
   <param name="p" type="{(x ↦ int)} → ((LI, ⇕), ∅)"/>
   <include>
      <call callee="p">
         <actualparam param="x"> 815 </actualparam>
      </call>
   </include>
</nsp>

<nsp name="D">
   <html>
      <head><title> Main Page </title></head>
      <body>
         <ul>
            <call callee="C">
               <actualparam param="p"> A </actualparam>
            </call>
            <call callee="C">
               <actualparam param="p"> B </actualparam>
            </call>
         </ul>
      </body>
   </html>
</nsp>
```

$\Gamma_{CONST} \equiv_{\texttt{DEF}} \{\texttt{815} \mapsto \texttt{int}\}$

$w_A \equiv_{\texttt{DEF}} \{(x \mapsto \texttt{array of int})\}$
$\Gamma_A \equiv_{\texttt{DEF}} \Gamma_{CONST} \cup \Delta_A \cup w_A$
$T_A \equiv_{\texttt{DEF}} w_A \to ((\circ, \Updownarrow), \emptyset)$
$\Delta_A \equiv_{\texttt{DEF}} \{(A \mapsto T_A)\}$

$w_B \equiv_{\texttt{DEF}} \{(x \mapsto \texttt{int})\}$
$\Gamma_B \equiv_{\texttt{DEF}} \Gamma_{CONST} \cup \Delta_B \cup w_B$
$T_B \equiv_{\texttt{DEF}} w_B \to ((LI, \Updownarrow), \emptyset)$
$\Delta_B \equiv_{\texttt{DEF}} \{(B \mapsto T_B)\}$

$w_C \equiv_{\texttt{DEF}} \{(p \mapsto T_B)\}$
$\Gamma_C \equiv_{\texttt{DEF}} \Gamma_{CONST} \cup \Delta_C \cup w_C$
$\Delta_C \equiv_{\texttt{DEF}} \big\{\big(C \mapsto \big(w_C \to ((LI, \Updownarrow), \emptyset)\big)\big)\big\}$

$\Gamma_D^{FINAL} \equiv_{\texttt{DEF}} \Gamma_{CONST} \cup \Delta_A \cup \Delta_B \cup \Delta_C$
$\Gamma_D \equiv_{\texttt{DEF}} \Gamma_D^{FINAL} \cup \Delta_D$
$\Delta_D \equiv_{\texttt{DEF}} \{(\ D \mapsto (\emptyset \to \Box)\ )\}$

I	$\Gamma_A \vdash \varepsilon : ((\circ, \Updownarrow), \emptyset)$	B.8
II	$\Gamma_A \vdash$ 03:$\emptyset \to ((\circ, \Updownarrow), \emptyset)$	I B.6
III	$\Gamma_A \vdash$ x:array of int	B.33
IV	$\Gamma_A \setminus w_A \vdash$ 02-03:T_A	III II B.4
V	$\Gamma_A \setminus w_A \vdash A : T_A$	B.33
VI	$\Gamma_{CONST}, \Delta_A \vdash$ 01-04:$\diamond$	V IV B.3
VII	$\Gamma_B \vdash$ x:int	B.33
VIII	$\Gamma_B \vdash$ 10:$((\bullet, \Updownarrow), \emptyset)$	VII B.26
IX	$\Gamma_B \vdash$ 09-11:$(LI, \Updownarrow), \emptyset)$	VIII B.11
X	$\Gamma_B \vdash$ 08-12:$\emptyset \to ((LI, \Updownarrow), \emptyset)$	IX B.6
XI	$\Gamma_B \setminus w_B \vdash$ 07-12:T_B	VII X B.4
XII	$\Gamma_B \setminus w_B \vdash B : T_B$	B.33
XIII	$\Gamma_{CONST}, \Delta_B \vdash$ 06-13:$\diamond$	XII XI B.3
XIV	$\Gamma_C \vdash$ "815":int	B.33
XV	$\Gamma_C \vdash$ 19:w_B	$\varepsilon_{\mathsf{act}}:\emptyset$ XIV B.17
XVI	$\Gamma_C \vdash$ p:T_B	B.33
XVII	$\Gamma_C \vdash$ 18-20:$((LI, \Updownarrow), \emptyset)$	XVI XV $w_B < w_B$ B.15
XVIII	$\Gamma_C \vdash$ 17-21:$\emptyset \to ((LI, \Updownarrow), \emptyset)$	XVII B.6
XIX	$\Gamma_C \setminus w_C \vdash$ 16-21:$w_C \to ((LI, \Updownarrow), \emptyset)$	XVI XVIII B.4
XX	$\Gamma_C \setminus w_C \vdash$ C:$w_C \to ((LI, \Updownarrow), \emptyset)$	B.33
XXI	$\Gamma_{CONST}, \Delta_C \vdash$ 15-22:$\diamond$	XX XIX B.3

XXII	$\Gamma_D \vdash \mathtt{A}{:}T_A$	B.33
XXIII	$\Gamma_D \vdash \mathtt{30}{:}\{(p \mapsto T_A)\}$	$\varepsilon_{\mathsf{act}}{:}\emptyset$ XXII B.17
XXIV	$\Gamma_D \vdash \mathtt{C}{:}w_C \rightarrow ((LI, \Updownarrow), \emptyset)$	B.33
XXV	$\vdash w_B < w_A$	array/record subtyping
XXVI	$\vdash ((\circ, \Updownarrow), \emptyset) < ((LI, \Updownarrow), \emptyset)$	layout subtyping
XXVII	$\vdash T_A < T_B$	XXV XXVI page subtyping
XXVIII	$\vdash \{\, p \mapsto T_A\} < w_C$	XXVII record subtyping
XXIX	$\Gamma_D \vdash \mathtt{29\text{-}31}{:}((LI, \Updownarrow), \emptyset)$	XXIV XXIII XXVIII B.15
XXX	$\Gamma_D \vdash \mathtt{B}{:}T_B$	B.33
XXXI	$\Gamma_D \vdash \mathtt{33}{:}w_C$	$\varepsilon_{\mathsf{act}}{:}\emptyset$ XXX B.17
XXXII	$\Gamma_D \vdash \mathtt{31\text{-}32}{:}((LI, \Updownarrow), \emptyset)$	XXIV XXXI $w_c < w_c$ B.15
XXXIII	$\Gamma_D \vdash (LI, \Updownarrow) \sqcup (LI, \Updownarrow) = (LI, \Updownarrow) \ \wedge\ \emptyset \odot \emptyset = \emptyset$	
XXXIV	$\Gamma_D \vdash \mathtt{29\text{-}34}{:}((LI, \Updownarrow), \emptyset)$	XXIX XXXII XXXIII B.9
XXXV	$\Gamma_D \vdash \mathtt{28\text{-}35}{:}((\bullet, \Updownarrow), \emptyset)$	XXXIV B.10
XXXVI	$\Gamma_D \vdash \mathtt{25\text{-}37}{:}\emptyset \rightarrow \Box$	XXXV B.5
XXXVII	$\Gamma_D \vdash \mathtt{D}{:}\emptyset \rightarrow \Box$	
XXXVIII	$\Gamma_D^{FINAL}, \Delta_D \vdash \mathtt{24\text{-}38}{:}\Diamond$	XXXVII XXXVI B.3
	$(dom\ \Delta_A) \cap (dom\ \Delta_B) = \emptyset$	
XXXIX	$\wedge\ \ ((dom\ \Gamma_{Const}) \lhd\ \Delta_B) = \emptyset = ((dom\ \Delta_A) \lhd\ \Gamma_{Const})$	
	$\wedge\ \ ((dom\ \Gamma_{Const}) \lhd\ \Delta_A) = \emptyset = ((dom\ \Delta_B) \lhd\ \Gamma_{Const})$	
XXXX	$\Gamma_{CONST}, \Delta_A \cup \Delta_B \vdash \mathtt{01\text{-}13}{:}\Diamond$	XXXIX VI XIII B.2
XXXXI	*analog of* XXXIX *concerning* $(\emptyset, \Delta_A \cup \Delta_B)$ *and* $(\emptyset, \Delta_D)$	
XXXXII	$\Gamma_{CONST}, \Delta_A \cup \Delta_B \cup \Delta_C \vdash \mathtt{01\text{-}22}{:}\Diamond$	XXXXI XXXX XXI B.2
	$(dom\ (\Delta_A \cup \Delta_B \cup \Delta_C)) \cap (dom\ \Delta_D) = \emptyset$	
XXXXIII	$\wedge\ \ (dom\ \Gamma_D^{FINAL}) \lhd (\Delta_A \cup \Delta_B \cup \Delta_C) = (dom(\Delta_A \cup \Delta_B \cup \Delta_C)) \lhd \Gamma_D^{FINAL}$	
	$\wedge\ \ (dom\ \Gamma_{CONST}) \lhd \Delta_D = (dom\ \Delta_D) \lhd \Gamma_{CONST}$	
XXXXIV	$\Gamma_{CONST}, \Delta_A \cup \Delta_B \cup \Delta_C \cup \Delta_D \ \vdash\ \mathtt{01\text{-}38}{:}\Diamond$	XXXXIII XXXXII XXXVIII B.2
XXXXV	$\Gamma_{CONST}, \Delta_A \cup \Delta_B \cup \Delta_C \cup \Delta_D \ \vdash\ \mathtt{01\text{-}38}{:}\surd$	XXXXIV ($\Gamma_{CONST} \in \mathbb{R}$) B.1

B.8 Example Type Derivation II

Listing B.2

```
<form callee="target">
  <code>
    if cond then
      while cond do
        </code>
          <object param="x">
            <input param="name" type="String">
            <object param="next">
              <input param="name" type="String">
            </object>
          </object>
        <code>
      ;
      </code>
        <object param="x">
          <input param="name" type="String">
          <object param="address">
            <input param="street" type="String">
            <input param="zip" type="int">
          </object>
        </object>
      <code>
    else
     </code>
       <select param="x">
          <option>
            <value> v </value>
            <label> l </label>
          </option>
        </select>
      <code>
  </code>
  <submit/>
</form>
```

Listing B.3

```
01 <nsp name="target">
02    <param name="x" type="array of
03                               μX.{name↦String,
04                                    address↦{street↦String,
05                                                 zip↦int},
06                                    next↦X
07                                  }"
08    />
09    <html>
10      ...
11    </html>
12 </nsp>
```

$w_{target} \equiv_{\mathtt{DEF}} \{\ x \mapsto \mathsf{array\ of}\ \mu X.\{name \mapsto \mathtt{String}, address \mapsto \{street \mapsto \mathtt{String}, zip \mapsto \mathtt{int}\}, next \mapsto X\}\ \}$

$\Gamma \equiv_{\mathtt{DEF}} \{\ cond \mapsto \mathtt{boolean}\ ,\ v \mapsto \mu X.\{name \mapsto \mathtt{String}, address \mapsto \{street \mapsto \mathtt{String}, zip \mapsto \mathtt{int}\}, next \mapsto X\}\ ,\ l \mapsto \mathtt{String}\ ,\ target \mapsto (w_{target} \rightarrow \Box)\ \}$

I	$\Gamma \vdash$ 09:$((\bullet,\Downarrow), \{name \mapsto \mathtt{String}\})$	B.22
II	$\Gamma \vdash$ 08-10:$((\bullet,\Downarrow), \{next \mapsto \{name \mapsto \mathtt{String}\}\})$	I B.19
III	$\Gamma \vdash$ 07:$((\bullet,\Downarrow), \{name \mapsto \mathtt{String}\})$	B.22
IV	$\Gamma \vdash$ 07-10:$((\bullet,\Downarrow), \{name \mapsto \mathtt{String}, next \mapsto \{name \mapsto \mathtt{String}\}\})$	I III B.9
V	$\Gamma \vdash$ 06-11:$((\bullet,\Downarrow), \{x \mapsto \{name \mapsto \mathtt{String}, next \mapsto \{name \mapsto \mathtt{String}\}\}\})$	IV B.19
VI	$\Gamma \vdash$ 05-12:$((\bullet,\Downarrow), \{x \mapsto \{name \mapsto \mathtt{String}, next \mapsto \{name \mapsto \mathtt{String}\}\}\})$	V B.28
VII	$\Gamma \vdash cond$:boolean	B.33
VIII	$\Gamma \vdash$ 04-12:$((\bullet,\Downarrow), \{x \mapsto \mathsf{array\ of}\ \{name \mapsto \mathtt{String}, next \mapsto \{name \mapsto \mathtt{String}\}\}\})$	VII VI B.31
IX	$\Gamma \vdash$ 19:$((\bullet,\Downarrow), \{zip \mapsto \mathtt{int}\})$	B.22
X	$\Gamma \vdash$ 18:$((\bullet,\Downarrow), \{street \mapsto \mathtt{String}\})$	B.22
XI	$\Gamma \vdash$ 18-19:$((\bullet,\Downarrow), \{street \mapsto \mathtt{String}, zip \mapsto \mathtt{int}\})$	IX X B.9
XII	$\Gamma \vdash$ 17-20:$((\bullet,\Downarrow), \{adress \mapsto \{street \mapsto \mathtt{String}, zip \mapsto \mathtt{int}\}\})$	XI B.19
XIII	$\Gamma \vdash$ 16:$((\bullet,\Downarrow), \{name \mapsto \mathtt{String}\})$	B.22
XIV	$\Gamma \vdash$ 16-20:$((\bullet,\Downarrow), \{name \mapsto \mathtt{String}, adress \mapsto \{street \mapsto \mathtt{String}, zip \mapsto \mathtt{int}\}\})$	XIII XII B.9
XV	$\Gamma \vdash$ 15-21:$((\bullet,\Downarrow), \{x \mapsto \{name \mapsto \mathtt{String}, adress \mapsto \{street \mapsto \mathtt{String}, zip \mapsto \mathtt{int}\}\}\})$	XIV B.19

XVI	$\Gamma \vdash$ 14-22:$((\bullet, \Downarrow), \{x \mapsto \{name \mapsto \texttt{String}, adress \mapsto \{street \mapsto \texttt{String}, zip \mapsto \texttt{int}\}\}\})$	XV B.28
XVII	$\Gamma \vdash$ 04-22:$((\bullet, \Downarrow), \{x \mapsto \texttt{array of } \{name \mapsto \texttt{String}, adress \mapsto \{street \mapsto \texttt{String}, zip \mapsto \texttt{int}\}, next \mapsto \{name \mapsto \texttt{String}\}\}$	VIII XVI B.29
XVIII	$\Gamma \vdash$ v:$\mu X.\{name \mapsto \texttt{String}, address \mapsto \{street \mapsto \texttt{String}, zip \mapsto \texttt{int}\}, next \mapsto X\}$	B.33
XIX	$\Gamma \vdash$ l:$String$	B.33
XX	$\Gamma \vdash$ 26-29:$((OP, \Updownarrow), \{x \mapsto \texttt{array of } \mu X.\{name \mapsto \texttt{String}, address \mapsto \{street \mapsto \texttt{String}, zip \mapsto \texttt{int}\}, next \mapsto X\}\})$	XVIII XIX B.25
XXI	$\Gamma \vdash$ 25-30:$((\bullet, \Downarrow), w_{target})$	XX B.24
XXII	$\Gamma \vdash$ 24-31:$((\bullet, \Downarrow), w_{target})$	XXI B.28
XXIII	$\Gamma \vdash$ 03-31:$((\bullet, \Downarrow), w_{target})$	XVII XXII B.30
XXIV	$\Gamma \vdash$ 02-32:$((\bullet, \Downarrow), w_{target})$	XVIII B.27
XXV	$\Gamma \vdash$ 33:$((\bullet, \Downarrow), \emptyset)$	B.21
XXVI	$\Gamma \vdash$ 02-33:$((\bullet, \Downarrow), w_{target})$	XXIV XXV B.9
XXVII	$\Gamma \vdash target : w_{target} \rightarrow \square$	B.33
XXVIII	$\Gamma \vdash$ 01-34:$((\bullet, \Uparrow), \emptyset)$	XXVII XXVI $\vdash w_{target} < w_{target}$ B.18

Appendix C

NSP Document Type Definitions

C.1 NSP Core Language XML DTD

```
<!ELEMENT nsp (param*,(html|include))>
<!ATTLIST nsp name CDATA #REQUIRED>

<!ELEMENT param EMPTY>
<!ATTLIST param name CDATA #REQUIRED>
<!ATTLIST param type CDATA #REQUIRED>

<!ELEMENT html (head,body)>
<!ELEMENT head (title)>
<!ELEMENT title (#CDATA)>

<!ELEMENT body %Dynamic;>
<!ELEMENT include %Dynamic;>

<!ENTITY % Dynamic "(%dynamic)*">
<!ENTITY % dynamic " #PCDATA | ul |li | table |tr | td | call
| form | object | hidden | submit | input | checkbox
| select | option | code | expression ">

<!ELEMENT ul %Dynamic;>
<!ELEMENT li %Dynamic;>
<!ELEMENT table %Dynamic;>
<!ELEMENT tr %Dynamic;>
<!ELEMENT td %Dynamic;>

<!ELEMENT call (actualparam)+>
```

```
<!ATTLIST call callee CDATA #REQUIRED>

<!ELEMENT actualparam (#PCDATA)>
<!ATTLIST actualparam param CDATA #REQUIRED>

<!ELEMENT form %Dynamic;>
<!ATTLIST form callee CDATA #REQUIRED>

<!ELEMENT object %Dynamic;>
<!ATTLIST object param CDATA #REQUIRED>

<!ELEMENT hidden (#PCDATA)>
<!ATTLIST hidden param CDATA #REQUIRED>

<!ELEMENT submit EMPTY>

<!ELEMENT input EMPTY>
<!ATTLIST input type CDATA #REQUIRED>
<!ATTLIST input param CDATA #REQUIRED>

<!ELEMENT checkbox EMPTY>
<!ATTLIST checkbox param CDATA #REQUIRED>

<!ELEMENT select %Dynamic;>
<!ATTLIST select param CDATA #REQUIRED>
<!ELEMENT option (value,label)>
<!ELEMENT value (#PCDATA)>
<!ELEMENT label (#PCDATA)*>

<!ELEMENT code %Dynamic;>

<!ELEMENT expression (#PCDATA)>
```

C.2 NSP Core User Interface Description Language SGML DTD

```
<!ELEMENT html - - (head,body)>
<!ELEMENT head - - (title)>
<!ELEMENT title - - (#PCDATA)*>
<!ELEMENT body - - %Hypertext>

<!ENTITY % Hypertext "(#PCDATA | ul | table | form)*">

<!ELEMENT ul - - (li)>
<!ELEMENT li - - %Hypertext>

<!ELEMENT table - - (tr)>
<!ELEMENT tr - - (td)>
<!ELEMENT td - - %Hypertext>

<!ELEMENT form - - %Hypertext -(a) -(form) +(INPUT|SELECT)>
<!ATTLIST form action CDATA #REQUIRED>

<!ELEMENT INPUT - - EMPTY>
<!ATTLIST INPUT
TYPE %InputType; #IMPLIED
NAME CDATA #IMPLIED
VALUE CDATA #IMPLIED
CHECKED (CHECKED) #IMPLIED
>

<!ENTITY % InputType "(TEXT | HIDDEN | SUBMIT | CHECKBOX | RADIO)">

<!ELEMENT SELECT - - (OPTION+) -(INPUT|SELECT)>

<!ELEMENT OPTION - - (#PCDATA)*>
<!ATTLIST OPTION VALUE CDATA #IMPLIED>
```

Appendix D

Notation

$\lhd$	Domain restriction (Z Notation). If R is a relation and S a set, then $S \lhd R$ is the set of all tuples (x, y) that belong to R whereas x must belong to S.
$⩤$	Domain anti-restriction (Z Notation). If R is a relation and S a set, then $S⩤R$ is the set of all tuples (x, y) that belong to R whereas x must not belong to S.
$(dom\ R)$	Domain of a relation or function (Z Notation).
$\mathbb{F}\, S$	Set of finite subsets of set S (Z Notation).
$⇻$	Finite Partial function.
$t \downarrow$	Defined (relations). $R(\pi_1, \pi_2) \downarrow$ is true iff $(\pi_1, \pi_2) \in R$.
$d \downarrow$	Defined (functions). $F(\pi_1) \downarrow$ is true iff $\pi_1 \in (dom\ F)$.
$t \uparrow$	Undefined (relations). $R(\pi_1, \pi_2) \uparrow$ is true iff $(\pi_1, \pi_2) \notin R$.
$d \uparrow$	Undefined (functions). $F(\pi_1) \uparrow$ is true iff $\pi_1 \notin (dom\ F)$.

Appendix E

Anlagen gemäß Promotionsordnung

Erklärung

Ich versichere, daß ich alle Hilfsmittel und Hilfen zur Erstellung der Dissertation in der vorliegenden Arbeit angegeben habe. Ich versichere, daß ich die vorliegende Dissertation auf Grundlage der angegebenen Hilfsmittel und Hilfen selbständig angefertigt habe.

Berlin, Oktober 2002, Dirk Draheim

Zusammenfassung

In der Dissertation wird eine statische Semantik für einen Skriptseiten-Ansatz ("Server Pages"-Ansatz) definiert. Das resultierende Typkorrektheitskonzept garantiert ein typsicheres Zusammenspiel von dynamisch generierten Web-Formularen und anvisierten Server-seitigen Software-Komponenten. Das resultierende Konzept einer Beschreibungskorrektheit sichert Gültigkeit von generierten Dokumenten in bezug auf eine vorgegebene Benutzerschnittstellen-Beschreibungssprache. Es werden das Konzept einer Beschreibungssprachen-basierten Unterstützung komplexer Nachrichtentypen, das Konzept des virtuellen Austauschs komplexer benutzerdefinierter Objekte über den Benutzeragenten, das Konzept der funktionalen Dekomposition von Skriptseiten und das Konzept von Skriptseiten höherer Ordnung vorgeschlagen und in den streng getypten Skriptseiten-Ansatz vollständig integriert.

Web-Applikationen erfahren eine zunehmende Verbreitung. Web-basierte Präsentationsschichten finden sich in verstärktem Maß als integraler Bestandteil von Unternehmensapplikationen. Diesem Umstand wird die Entwicklung einer großen Anzahl diverser Web-Technologien gerecht. Web-Applikation und Web-Technologie sind Gegenstand aktueller Forschung. Skriptseiten stellen in diesem Bereich einen neusten Stand der Technik dar. Die Beiträge der vorliegenden Arbeit zielen auf Robustheit, Wartbarkeit und Wiederverwendbarkeit von Skriptseiten-basierten Applikationen. Die erzielten Resultate sind programmiersprachenunabhängig, konkrete Programmiersprachen können semantik-erhaltend mit den gefundenen Konzepten verschmolzen werden.

In der Dissertation wird zunächst eine abstrakte Charakterisierung von Skriptseiten als streng getypte Dialogmethoden erarbeitet. Auf dieser Basis werden Kodierungsrichtlinien und -regeln definiert, die zusammen die gewünschte Typkorrektheit und Beschreibungskorrektheit informell begründen. Die statische Semantik des Skriptseiten-Ansatzes wird als "Per Martin-Löf"-artiges Typsystem bezüglich der Verschmelzung mit einer minimalen imperativen Programmiersprache und einem ausreichend komplexen equi-rekursiven Typsystem formalisiert. Die Entwurfsentscheidungen des Ansatzes werden durch die Analyse einer verbreiteten Software-Architektur von Web-Applikationen zusätzlich motiviert. Die operationelle Semantik einer konkreten Sprachverschmelzung wird als Transformation in eine bestehende Technologie ausformuliert. Außerdem wird ein rechnergestütztes Entwicklungswerkzeug zur statischen Analyse ererbter Skriptseiten-basierter Präsentationsschichten vorgestellt. Die Semantik dieses Werkzeugs wird als Pseudo-Auswertung formal definiert.

Der erarbeitete Skriptseiten-Ansatz ist in einen Gesamtansatz zur Modellierung und Entwicklung formularbasierter Systeme integriert. Ein Fallbeispiel demonstriert die relative Strukturbruchlosigkeit des Gesamtansatzes.

Curriculum Vitae

Dirk Draheim
Carstennstr. 30c
12055 Berlin

	geboren 02.06.1969, Berlin-Spandau
1975-1981	Grundschule am Birkenhain, Berlin-Spandau
1981-1988	Herder-Gymnasium, Berlin-Charlottenburg
1988-1994	Studium Informatik, Technische Universität Berlin
1998	Lehrauftrag, Software-Praktikum, Institut für Informatik, Freie Universität Berlin
1999	Lehrauftrag, Seminar Fortgeschrittene Aspekte der Semantik von Programmiersprachen, Institut für Informatik, Freie Universität Berlin
1999	Lehrauftrag, Softwaretechnik, Institut für Informatik, Freie Universität Berlin
2000	Lehrauftrag, Softwaretechnik, Institut für Informatik, Freie Universität Berlin
2001	Lehrauftrag, Application-Server-gestützte Verwaltungssysteme, Institut für Informatik, Freie Universität Berlin
2001	Lehrauftrag, Semantik von Programmiersprachen, Institut für Informatik, Freie Universität Berlin
2002	Lehrauftrag, Domain-Theorie, Institut für Informatik, Freie Universität Berlin
2000-2002	Wissenschaftlicher Mitarbeiter von Professor Elfriede Fehr, Arbeitsgruppe Programmiersprachen und Rechnerarchitektur, Institut für Informatik, Freie Universität Berlin

Bibliography

[1] Martin Abadi, Luca Cardelli. A Theory of Primitive Objects - Untyped and First-Order Systems. Information and Computation, 125(2), pp.78-102, 1996. Earlier version appeared in TACS '94 proceedings, LNCS 789, 1994.

[2] Martin Abadi, Luca Cardelli. A Theory Of Objects. Springer, 1996.

[3] Roberto M. Amadio, Luca Cardelli: Subtyping Recursive Types. In: ACM Transactions on Languages and Systems, vol. 15, no.4., pp. 575-631, September 1993.

[4] Kirby W. Angell. Python Server Pages (PSP), Part I. Dr. Dobbs Journal, January 2000.

[5] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia. Web site reengineering using RMM. In: Proc. of the International Workshop on Web Site Evolution, pages 9-16, March 2000.

[6] David Atkins, Thomas Ball et al. Integrated Web and Telephone Service Creation. Bell Labs Technical Journal, 2(1), Winter 1997.

[7] David Atkins, Thomas Ball, Michael Benedikt, Glenn Bruns, Kenneth Cox, et.al. Experience with a Domain Specific Language for Form-based Services. Software Production Research Department, Bell Laboratories, Lucent Technologies. 1997.

[8] David Atkins, Thomas Ball, Glenn Bruns, and Kenneth Cox. Mawl: a domain-specific language for form-based services. In: IEEE Transactions on Software Engineering, June 1999.

[9] Attardi, G., Cisternino, A.: Reflection Support by Means of Template Metaprogramming. In: LNCS 2186, pp. 118. Springer, 2001.

[10] H.P. Barendregt: Lambda Calculi with Types. In: S.Abramsky, D.V. Gabbay, T.S.E. Maibaum (eds.): Handbook of Logic in Computer Science, vol.2., pp.118-331. Clarendon Press, 1992.

[11] Beck, K.: Extreme Programming Explained - Embrace Change. Addison-Wesley, 2000

[12] T. Berners-Lee, Dan Connolly. Hypertext Markup Language - 2.0, RFC 1866. Network Working Group. November 1995

[13] Grady Booch, James Rumbaugh, Ivar Jacobson (The Three Amigos). The Unified Modeling Language User Guide. Addison-Wesley, 1999

[14] Don Box et.al. Simple Object Access Protocol (SOAP) 1.1 - W3C Note, May 2000.
http://www.w3.org/TR/SOAP/

[15] Claus Brabrand, Anders Møller, Anders Sandholm, and Michael I. Schwartzbach. A runtime system for interactive Web services. Computer Networks, 31:1391-1401, 1999. Also in Proceedings of the Eighth International World Wide Web Conference.

[16] C. Brabrand, A. Møller, M. Ricky, M.I. Schwartzbach, "Powerforms: Declarative client-side form field validation", World Wide Web Journal, 3(4), 2000.

[17] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. Static validation of dynamically generated HTML. In: Proceedings of Workshop on Program Analysis for Software Tools and Engineering. ACM, 2001.

[18] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. The bigwig project. ACM Transactions on Internet Technology, to appear.

[19] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler. Extensible Markup Language (XML) 1.0 (Second Edition). World Wide Web Consortium, 2000.

[20] Brown et. al. Professional JSP, 2nd edition. Wrox Press, April 2001

[21] Kenneth A. Bruffee: Collaborative Learning: Higher Education, Interdependence, and the Authority of Knowledge. Johns Hopkins University Press, 1993.

[22] David Budgen, Sarah Burgees: A Simple Tool for Temporal Indexing of Hypertext Documents. In: Computer 31, pp.52-53. IEEE, December 1998.

[23] Mark H. Butler. Current Technologies for Device Independence - HP Labs Technical Report HPL-2001-83. Hewlett-Packard Company, April 2001.

[24] Daniel Cabeza, Manuel Hermenegildo. The PiLLoW Web Programming Library Reference Manual. The CLIP Group, School of Computer Science, Technical University of Madrdid, 2000.
http://citeseer.nj.nec.com/cabeza00pillow.html

[25] Daniel Cabeza, Manuel Hermenegildo. WWW Programming using Computational Logic Systems (and the PILLOW/CIAO Library). In: Proc. of the Workshop on Logic Programming and the WWW, in conjunction with WWW6, April 1997.

[26] Luca Cardelli, Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. In: Computing Surveys, Vol. 17, No. 4, ACM 1985.

[27] Luca Cardelli. Type systems. In: Handbook of Computer Science and Engineering. CRC Press, 1997

[28] Felice Cardone and Mario Coppo. Type Inference with Recursive Types: Syntax and Semantics. Information and Computation, 1990.

[29] Elliot J. Chikofsky and James H. Cross, II. Reverse engineering and design recovery: A taxonomy. IEEE Software, pp. 13-17, January 1990.

[30] Jean-François Chosson: L'entraînement mental. Le Seuil, 1975.

[31] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Extending Java for High-Level Web Service Development. Technical Report RS-02-11, BRICS, Department of Computer Science, University of Aarhus, March 2002.

[32] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Static Analysis for Dynamic XML. Proceedings of Workhsop PLAN-X - Programming Language Technologies for XML, October 2002.

[33] Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. Ariba, International Business Machines Corporation, Microsoft. July 2001.
http://www.w3.org/TR/wsdl.

[34] James Clark. XSL Transformations (XSLT) - W3C Recommendation. World wide Web Consortium, November 1999.
http://www.w3.org/TR/xslt

[35] James Clark, Steve DeRose. XML Path Language (XPath) Version 1.0 - W3C Recommendation 16. World Wide Web Consortium, November 1999

[36] Peter Coad. Object Models - Strategies, Patterns, and Applications. Yourdon Press, A Prentice Hall Company, 1995.

[37] Peter Coad, David North, and Mark Mayfield. Strategies and Patterns Handbook. Object International, Inc. 1997. No publisher given.
http://thecoadletter.com/download/objectmodels/objectmodelshandbook22.pdf

[38] Ken A L Coar, D.R.T Robinson The WWW Common Gateway Interface - Version 1.1, Internet-draft, June 1999
http://cgi-spec.golux.com/draft-coar-cgi-v11-03.txt

[39] Cocoon. http://xml.apache.org/cocoon/

[40] Daniel W. Connolly Document Type Definition for the HyperText Markup Language, level 2. World Wide Web Consortium, 1995.

[41] Steve Cook and John Daniels. Designing Object Systems: Object-Oriented Modeling with Syntropy. Prentice Hall, 1994.

[42] Bruno Courcelle. Fundamental Properties of Infinite Trees. In: Theoretical Computer Science 25, pp.95-169. Norh-Holland Publishing Company, 1983.

[43] Patrick Cousot: Program Analysis: The Abstract Interpretation Perspective., SIGPLAN Notices, Volume 32, pp. 73-76, 1997.

[44] K. Cox, T. Ball, and J. C. Ramming. Lunchbot: A tale of two ways to program web services. Technical Report BL0112650-960216-06TM, AT&T Bell Laboratories, 1996.

[45] Crary, K., R. Harper and S. Puri. What is a recursive module? In: Proc. ACM Conference on Programming Language Design and Implementation, pages 50–63, May 1999.

[46] D. Crocker P. Overell (editors). Augmented BNF for Syntax Specifications: ABNF. Dequest for Comments: 2234. Network Working Group, November 1997.

[47] Czarnecki, K., Eisenecker, U. "Generative Programming - Methods, Tools, and Applications". Addison Wesley Publishers, 2000.

[48] James Duncan Davidson, Suzanne Ahmed. Java Servlet Specification, v2.1a. Sun Press, 1999

[49] James Duncan Davidson, Danny Coward. Java Servlet Specification, v2.2. Sun Press, 1999

[50] Davis, M.: Struts, an open-source MVC implementation.
IBM developerWorks, February 2001

[51] Pierre Delisle. Java Server Pages Standard Tag Library - version 1.0. Sun Microsystems, June 2002.

[52] Steve DeRose, Eve Maler, David Orchard. XML Linking Language (XLink) Version 1.0 - W3C Recommendation. World Wide Web Consortium, June 2001.
http://www.w3.org/TR/xlink/

[53] Arie van Deursen, Paul Klint, and Joost Visser. Domain-Specific Languages: An Annotated Bibliography. ACM SIGPLAN Notices, 35(6):26–36, June 2000. (p 72)

[54] Edsger Wybe Dijkstra. Go to statement considered harmful. Communications of the ACM, 11(3):147-148, 1968.

[55] Draheim, D.: Integration von Polymorphismus und Subtypen für den Pi-Kalkül. In: Vortragsband zum 18. Workshop der GI-Fachgruppe 2.1.4. Christian-Albrechts-Universität zu Kiel, Bericht Nr. 2018, Oktober 2001.

[56] Draheim, D., Weber, G.: Specification and Generation of JSP Dialogues with Gently. In: Proceedings of NetObjectDays 2001, tranSIT, ISBN 3-00-008419-3, September 2001.

[57] Draheim, D., Weber, G.: Strong Complex Typed Dialogue-Oriented Server Pages. Technical Report B-02-05. Institute of Computer Science, Freie Universität Berlin, March 2002.

[58] Draheim, D., Weber, G.: An Introduction to Form Storyboarding. Technical Report B-02-06. Institute of Computer Science, Freie Universität Berlin, March 2002.

[59] Draheim, D., Weber, G.: An Overview of state-of-the-art Architectures for Active Web Sites. Technical Report B-02-07. Institute of Computer Science, Freie Universität Berlin, March 2002.

[60] Draheim, D., Weber, G.: Form Charts and Dialogue Constraints. Technical Report B-02-08. Institute of Computer Science, Freie Universität Berlin, March 2002.

[61] Draheim, D., Weber, G.: An Introduction to State History Diagrams. Technical Report B-02-09, Institute of Computer Science, Freie Universität Berlin, March 2002.

[62] Draheim, D., Weber, G.: Strongly Typed Server Pages. In: Proceedings of The Fifth Workshop on Next Generation Information Technologies and Systems, LNCS. Springer-Verlag, June 2002.

[63] Draheim, D., Fehr, E., and Weber, G.: The Definition of the NSP Type System. Technical Report B-02-11, Institute of Computer Science, Freie Universität Berlin, October 2002.

[64] Draheim, D., Lutteroth, C. and Weber, G.: An Analytical Comparison of Generative Programming Technologies. In: Proceedings of the 19. Workshop GI Working Group 2.1.4. Technical Report at Christian-Albrechts-University of Kiel, to appear.

[65] Draheim, D.: Learning Software Engineering with EASE. In: Proceedings of SECIII Open IFIP Conference on Social, Ethical, Cognitive Issues of Informatics and ICT. Kluwer Academic Publishers, to appear.

[66] Draheim, D. , Weber, G., Yassin, N.: The JSPick Project - A Java Server Pages Analyzer. October 2002.
http://www.inf.fu-berlin.de/projects/jspick/

[67] Draheim, D., Weber, G.: - A Specification driven Generator for Java Server Pages.
http://www.inf.fu-berlin.de/projects/gently/

[68] Micah Dubinko, Sebastian Schnitzenbaumer, Malte Wedel, Dave Raggett. XForms Requirements - W3C Working Draft. World Wide Web Consortium, April 2001.
http://www.w3.org/TR/xhtml-forms-req

[69] Micah Dubinko, Leigh L. Klotz, Roland Merrick, T. V. Raman. XForms 1.0. - W3C Working Draft. World Wide Web Consortium. August 2002.
http://www.w3.org/TR/xforms/.

[70] ECMA - European Computer Manufacturer's Association. Standard ECMA-262 - ECMAScript Language Specification. ECMA Standardizing Information and Communication Systems, 1999.

[71] Elfriede Fehr. Semantik von Programmiersprachen. Springer-Verlag, 1989.

[72] R. T. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol - HTTP/1.1. RFC 2616. IETF - Network Working Group, The Internet Society, June 1999.

[73] David Flanagan. JavaScript: The Definitive Guide. O'Reilly, 2002.

[74] Martin Fowler: UML Distilled - Applying the Standard Object Modeling Language. Addison Wesley, 1997.

[75] Gamma et al. Design Patterns. Addison Wesley, 1995

[76] Vladimir Gapayev, Michael Y. Levin, Benjamin C. Pierce. Recursive Subtyping Revealed. In: International Conference on Functional Programming, 2000. To appear in Journal of Functional Programming

[77] Gartner Group. Survey Results: The Real Cost of E-Commerce Sites. Gartner Group, 1999.

[78] Andy Gill. HTML combinators, version 2.0.
http://www.cse.ogi.edu/ andy/html/intro.htm, 2002.

[79] J. Gosling, B. Joy, G. Steele and G.Bracha: The Java Language Specification. Addison-Wesley,1996).

[80] Mark Grand. Patterns in Java - A Catalog of Reusable Design Patterns. John Wiley & Sons, 1998.

[81] P. Graunke, S. Krishnamurthi, S. van der Hoeven, and M. Felleisen. Programming the web with high-level programming languages. In European Symposium on Programming, 2001.

[82] Greenberg, S.: Computer-supported Cooperative Work and Groupware, London, Academic Press Ltd, 1991

[83] Gunter, C.A.: Semantics of Programming Languages - Structures and Techniques. The MIT Press, 1992.

[84] Hugo Haas, David Orchard. Web Services Architecture Usage Scenarios - W3C Working Draft. World Wide Web Consortium, July 2002. http://www.w3.org/TR/ws-arch-scenarios/

[85] Graham Hamilton (editor). Java Beans version 1.0.1. Sun Microsystems, July 1997.

[86] Michael Hanus. Server side Web scripting in Curry. In Workshop on (Constraint) Logic Programming and Software Engineering (LPSE2000), London, July 2000.

[87] Michael Hanus : High-level server side Web scripting in Curry . In: Practical Aspects of Declarative Languages, Proceedings of the Third International Workshop, PADL'01. Lecture Notes in Computer Science, Las Vegas, NV, USA, SpringerVerlag, 2001.

[88] Ahmed E. Hassan, Richard C. Holt. Towards a Better Understanding of Web Applications, Proceedings of WSE 2001: International Workshop on Web Site Evolution, Florence, Italy, Nov 10, 2001.

[89] Ahmed E. Hassan, Richard C. Holt. Architecture Recovery of Web Applications, Proceedings of ICSE 2002: International Conference on Software Engineering, Orlando, Florida, 19-25 May 2002.

[90] Ahmed E. Hassan, Richard C. Holt. A Visual Architectural Approach to Maintaining Web Applications, Annals of Software Engineering - Volume 16 - Special Volume on Software Visualization, 2003.

[91] Ian Hayes. Specification Case Studies. Prentice Hall, 1993.

[92] Enrico Heine. Vergleichende Analyse verbreiteter und neuster Software-Projekt-Management-Werkzeuge unter besonderer Berücksichtigung studentischer Software-Projekte. Seminar paper. Institute of Computer Science, Freie Universität Berlin, August 2002.

[93] Enrico Heine. Ein kombiniertes Projekt-Management und CSCW-Werkzeug für den Software-Prozeß EASE. Diploma Thesis. Institute of Computer Science, Freie Universität Berlin, to appear.

[94] John Hughes. Generalising Monads to Arrows. Science of computer programming, vol.37, pp.37-111, 2000.

[95] Institute of Electrical and Electronics Engineers. IEEE Standard 830-1993, Recommended Practice for Software Requirements Specifications, Software Engineering Standards Committee of the IEEE Computer Society, New York, 1993.

[96] International Standardization Organisation. International Standard ISO/IEC 16262 - ECMAScript: A general purpose, cross-platform programming language.. ISO,1998.

[97] International Standardization Organisation. International Standard ISO/ICE 14977. Syntactic metalanguage - Extended BNF. ISO,1996.

[98] International Standardization Organisation. ISO 8879. Information Processing – Text and Office Systems - Standard Generalized Markup Language (SGML). ISO,1986.

[99] International Organization for Standardization. ISO/IEC 10744-1992 (E): Information technology-Hypermedia/Time-based Structuring Language (HyTime). International Organization for Standardization, 1996. http://www.ornl.gov/sgml/wg8/docs/n1920/html/n1920.html

[100] J. Jensen. Generation of machinecode in ALGOL compilers. BIT - Nordisk Tidskrift for Informations-Behandling. Volume 5, pp.235-245, 1965.

[101] Trevor Jim and Jens Palsberg. Type inference in systems of recursive types with subtyping. Manuscript, 1999.

[102] Nicholas Kassem and the Enterprise Team. Designing Enterprise Applications with the Java 2 Platform, Enterprise Edition. Sun Microsystems, 2000

[103] Pekka Kilpeläinen, Derick Wood. SGML and Exceptions, Department of Computer Science, University of Helsinki, Technical Report HKUST-CS96-03, 1996.

[104] Krasner, G.E., Pope, S.T. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. In: Journal of Object-Oriented Programming, August/September 1988 26-49

[105] D. Kristol, L. Montulli. HTTP State Management Mechanism. RFC 2109. Network Working Group. February 1997.

[106] Ladd, D.A., and Ramming, J.C. Programming the Web: An application-oriented language for hypermedia services. In 4th International World Wide Web Conference, 1995.

[107] Craig Larman. Applying UML and Patterns. Prentice Hall, 1998

[108] Rasmus Lerdorf, Kevin Tatroe. Programming PHP. O'Reilly, 2002.

[109] Michael R. Levy. Web Programming in Guide. Software - Practice and Experience, vol. 28(15), pp. 1581-1603, December 1998.

[110] Jesse Liberty, Dan Hurwitz. Programming APS.NET. O'Reilly, 2002.

[111] Barbara Liskov. Data Abstraction and Hierarchy. SIGPLAN Notices. 23(5), May 1988.

[112] G.A.Di Lucca, A. R. Fasolino, F. Pace, P. Tramontana, U. de Carlini. WARE: A Tool for the Reverse Engineering of Web Applications. In: Sixth European Conference on Software Maintenance and Reengineering. IEEE, 2002.

[113] T.J.Marlowe and B.G.Ryder. Properties of data flow frameworks - A unified model. Acta Informatica 28, pp. 121-163, 1990.

[114] Jonathan Marsh, David Orchard. XML Inclusions (XInclude) Version 1.0 - W3C Working Draft. World Wide Web Consortium, May 2001. http://www.w3.org/TR/2001/WD-xinclude-20010516/

[115] Per Martin-Löf. Constructive Mathematics and Computer Programming. In Logic, Methodology and Philosophy of Science, VI, 1979, pp. 153-175. North-Holland, 1982.

[116] Per Martin-Löf. Intuistionistic Type-Theory. Bibliopolis, Napoli, 1984.

[117] Stefano Mazzocchi, Ricardo Rocha. Extensible Server Pages. The Apache Software Foundation, 2001. http://xml.apache.org/cocoon1/wd-xsp.html

[118] Scott McGlashan et al. Voice Extensible Markup Language (VoiceXML) Version 2.0, W3C Working Draft, October 2001.

[119] Erik Meijer, Daan Leijen, James Hook: Client-Side Web Scripting with HaskellScript. In: Practical Aspects of Declarative Languages (PADL), LNCS 1551, 1pp. 96-210, 1999.

[120] Erik Meijer. Server-side Scripting in Haskell. Journal of Functional Programming, 2000.

[121] Erik Meijer and Mark Shields. XMλ - A Functional Language for Constructing and Manipulating XML Documents. http://www.cse.ogi.edu/∼mbs, Draft, 2000.

[122] Erik Meijer, Danny van Velzen. Haskell Server Pages - Functional Programming and the Battle for the Middle Tier. Electronic Notes in Theoretical Computer Science 41, No.1, Elsevier Science, 2001.

[123] Bertrand Meyer. Applying "design by contract". IEEE Computer, 25(10):40–51, October 1992

[124] Bertrand Meyer. Design by Contract. In: Dino Mandroli, Bertrand Meyer (editors). Advances in Object-Oriented Software Engineering. Prentice Hall, 1992.

[125] Microsoft Developer Network. Official Guidelines for User Interface Developers and Designers. Microsoft Corporation, 2002. http://msdn.microsoft.com/

[126] Mordani, R.; Davidson, J.D.; and Boag, S. Java API for XML Processing Specification, v1.1. Sun Microsystems, 2001.

[127] Yassin Naciri. JSPICK - Ein Reverse Engineering Tool für JSP-basierte Präsentationsschichten. Diploma Thesis. Freie Universität Berlin, November 2002.

[128] Peter Naur: The Design of the GIER ALGOL Compiler - Nordisk Tidskrift for Informations-Behandling. Volume 3, pp.124-140 and 145-166, 1963.

[129] Peter Naur: Checking of operand types in ALGOL compilers. BIT - Nordisk Tidskrift for Informations-Behandling. Volume 5, pp.151-163, 1965.

[130] Netscape Corporation. DevEdge Online Archive. Sample Code for Form Validation.
http://developer.netscape.com/docs/examples/javascript/formval/overview.html

[131] Flemming Nielson, Hanne Nielson, Chris Hankin: Principles of Program Analysis. Springer-Verlag, 1999.

[132] Bengt Nordström, Kent Peterson, Jan M. Smith: Programming in Martin-Löfs Type Theory. The International Series of Monographs on Computer Science. Clarendon Press, 1990.

[133] Kurt Nørmark, "Programming World Wide Web Pages in Scheme", Sigplan Notices, Vol. 34, No. 12, pp. 37–46, December 1999.

[134] Kurt Nørmark, "Using Lisp as a markup language—The LAML approach". In European Lisp User Group Meeting, Franz Inc., 1999.

[135] Kurt Nørmark. "Programmatic WWW authoring using Scheme and LAML". The Eleventh International World Wide Web Conferene 2002, March 2002.

[136] Open Source Development Network. sourceforge.net - Breaking Down the Barriers to Open Source Development. Open Source Development Network, 2002.
http://sourceforge.net/

[137] John K. Ousterhout. Tcl and the Tk Toolkit. Addison-Wesley, 1994.

[138] John K. Ousterhout. Scripting: Higher-Level Programming for the 21st Century. Computer, 31(3):23–30, 1998.

[139] Palsberg, J., and O'Keefe, P. A type system equivalent to flow analysis. In Proceedings of the ACM SIGPLAN '95 Conference on Principles of Programming Languages, pp. 367-378, 1995.

[140] Jens Palsberg and Michael I. Schwartzbach. Safety analysis versus type inference. Information and Computation, 118(1), pp.128-141, 1995.

[141] Terrence John Parr: Obtaining Practical Variants Of LL(k) And LR(k) For k-1 By Splitting The Atomic k-Tuple, Ph.D. Dissertation, School of Electrical Engineering, Purdue University, 1993.

[142] Terence Parr, Ric Klaren. ANTLR - Complete Language Translation Solutions. JGuru, 2002.
http://www.antlr.org/

[143] Eduardo Pelegri-Llopart, Larry Cable. Java Server Pages Specification, v.1.1. Sun Press, 1999

[144] Simon Peyton Jones and John (editors). Report on the programming language Haskell 98. Technical Report YALEU/DCS/RR-1106, Yale University, CS Dept., February 1999.

[145] Benjamin C. Pierce: Types and Programming Languages. MIT Press, 2002.

[146] Michael E. Porter. Strategy and the Internet. Harvard Business Review, March 2001, pp. 63-78, 2001.

[147] Shelley Powers. Developing ASP Components. O'Reilly, 1999.

[148] Christian Queinnec. "The Influence of Browsers on Evaluators or, Continuations to Program Web Servers", International Conference on Functional Programming, ACM. Montral, Canada, Nov. 2000.

[149] Evan Quinn. Application Server Market Share - A Different Angel. Hurwitz Balanced View Bulletin. Hurwitz Group Inc, December 2001.

[150] Dave Raggett, Arnaud Le Hors, Ian Jacobs. HTML 4.01 Specification - W3C Recommendation. World Wide Web Consortium, December 1999.
http://www.w3.org/TR/html401/

[151] J. Christopher Ramming. PML: A Language Interface to Distributed Voice-Response Units. ICCL Workshop: Internet Programming Languages. LNCS 1686,pp. 97-112, 1999.

[152] Eric Raymond. The Cathedral & the Bazaar - Musings on Linux and Open Source by an Accidental Revolutionary. O'Reilly, 1999.

[153] Eric Raymond (editor). The Jargon file, version 4.3.1.
http://www.tuxedo.org/jargon/,June 2001.

[154] Revans, R.W.: What is Action Learning ? In: The Journal of Management Development, vol. 1, no. 3. MCB Publications, 1982; pp. 64-75.

[155] Rosenfeld, L.; and Morville, P. Information Architecture for the World Wide Web. O'Reilly, 1998.

[156] James Rumbaugh, Ivar Jacobson, Grady Booch. The Unified Modeling Language - Reference Manual. Addison-Wesley, 1999

[157] Sandholm, A., Schwartzbach, M.I.: A type system for dynamic web documents . In Reps, T., ed.: Proc. 27th Annual ACM Symposium on Principles of Programming Languages, pp. 290-301, ACM Press, 2000.

[158] Davide Sangiorgi, David Walker. The π-calculus - A Theory of Mobile Processes. Cambridge University Press, 2001.

[159] Schaps, G.L. Compiler Construction with ANTLR and Java. In Dr. Dobb's Journal, March1999.

[160] Tatsurou Sekiguchi and Akinori Yonezawa. A complete type inference system for subtyped recursive types . In Proc. Theoretical Aspects of Computer Software, volume 789 of Lecture Notes in Computer Science, pages 667–686. Springer-Verlag, 1994.

[161] Seng Wai Loke, Andrew Davison: "Logic Programming with the World-Wide Web", the Proceedings of the 7th ACM Conference on Hypertext, ACM Press, pp. 235-245, 1996.

[162] Shields, M., and Meijer, E. Type-indexed rows . In Proceedings of the 28th Annual ACM SIGPLANSIGACT Symposium on Principles of Programming Languages (POPL'01), ACM Press, pp. 261-275, 2001.

[163] J.M. Spivey. The Z Notation. Prentice Hall, 1992.

[164] Sun Microsystems. Java Remote Method Invocation Specification, revision 1.50. Sun Microsystems, October 1998.

[165] Sun Microsystems. Java Object Serialization Specification, revision 1.43. Sun Microsystems, November 1998.

[166] Sun Microsystems. JavaTM 2 Platform, Standard Edition, version 1.4.1, API Specification. Sun Microsystems, 2002.
http://java.sun.com/j2se/1.4.1/docs/api/

[167] Sun Microsystems. Sun ONE Application Framework (JATO).
http://developer.iplanet.com/tech/appserver/framework/index.jsp

[168] Peter Thiemann. "Modeling HTML in Haskell". Practical Applications of Declarative Programming, PADL '00. volume 1753 of Lecture Notes in Computer Science. January 2000.

[169] Peter Thiemann. "A typed representation for HTML and XML documents in Haskell". February 2001.

[170] Peter Thiemann. "Wash/CGI: Server-side Web Scripting with Sessions and Typed, Compositional Forms". Practical Aspects of Declarative Languages (PADL'02). January 2002.

[171] S. Tucker Taft, Robert A. Duff, and T. Taft (Editors): Ada 95 Reference Manual: Language and Standard Libraries. International Standard Iso/Iec 8652:1995(E). Lecture Notes in Computer Science 1246. Springer, 1995.

[172] The W3C HTML working group. XHTML 1.0 The Extensible HyperText Markup Language.
http://www.w3.org/TR/xhtml1/. W3C, 2000.

[173] The W3C HTML working group. Extensible HTML version 1.0 Strict DTD.
http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd W3C, 2000.

[174] L. Wall, T. Christiansen, and J. Orwant. Programming Perl. O'Reilly, 2000.

[175] Wafer - Web Application Research Project.
http://www.waferproject.org/index.html

[176] William M. Waite, Gerhard Goos. Compiler Construction. Springer-Verlag, 1984

[177] Gregory H. Watson: Strategic benchmarking - How to rate your company's performance against the world's best. John Wiliey, 1993.

[178] Gerald Weber. Semantics and Pragmatics of Form-Oriented Analysis. Dissertation. Freie Universität Berlin, October 2002.

[179] Webmacro. http://www.webmacro.org/, 2002.

[180] M. Wallace and C. Runciman. Haskell and XML: Generic combinators or typebased translation? ACM SIGPLAN Notices, 34(9):148-159, Sept. 1999. Proceedings of ICFP'99.

[181] J. Warmer and S. van Egmond. The implementation of the Amsterdam SGML parser. Electronic Publishing, 2(2):65–90, July 1989.

[182] Warmer, J.; and Kleppe, A.G. The Object Constraint Language. Addison-Wesley, 1999.

[183] Eoin Whelan, Fergal McGrath. A Study of the Total Life Cycle Costs of an E-Commerce Investment. Evaluation and Programm Planning, Vol. 125, pp. 191-196. Pergamon. Elsevier Science, 2002.

[184] D. Wood. Standard generalized markup language: Mathematical and philosophical issues . In J. van Leeuwen, editor, Computer Science Today. Recent Trends and Developments, volume 1000 of Lecture Notes in Computer Science, pages 344-365. Springer-Verlag, 1995. 60

[185] J.B. Wordsworth. Software Developement with Z - A Practical Approach to Formal Methods in Software Engineering. Addison Wesley, 1992.

[186] White, S. et al. JDBC API Tutorial and Reference. Addison-Wesley, 1999.

[187] Niklaus Wirth: Programming in MODULA-2, 3rd Edition. Springer-Verlag, 1985.