Strongly Complex Typed, Dialogue-Oriented Server Pages

Technical Report B-02-05

Dirk Draheim and Gerald Weber
Institute of Computer Science
Free University Berlin
email: {draheim,weber}@inf.fu-berlin.de

March 2002

Abstract

We present NSP, a new, statically typed server pages technology. NSP supports arrays and user defined types within input forms. It is the result of designing server pages from scratch. It addresses the needs of server pages developers who want to build cleaner, better reusable, more stable systems.

Contents

1 Introduction **3**

2 Motivation **4**

3 A First Example **5**

4 Language Description **6**
4.1 Top-level Structure 6
4.2 Type System 6
4.2.1 Input Capabilities 7
4.2.2 Forms and Links 7
4.2.3 Arrays 9
4.2.4 User Defined Types 9
4.3 Functional Decomposition 11

5 Reverse Engineering of User Dialogues with Angie **12**

6 Related Work **12**
6.1 Web Services contra HTML/HTTP User Dialogues 12
6.2 Web Application Frameworks 12
6.3 The "System Calls User" Approach 13

7 Conclusion **13**

1 Introduction

Next Server Pages (NSP) is a programming model for system dialogues based on dynamically generated pages, like today's web interfaces. In the following we summarize NSP's original contributions to server pages development.

- Strong type system
  - Parameterized pages. A web application is accessed through a set of single pages, which together comprise the system interface. NSP pages are remote methods that send HTML output back to the caller.
  - Static type checker. NSP defines a strong type system which incorporates NSP pages that are called across the net. A new widget set is supporting technology for it. At deployment time the system interface is statically checked as a whole against sophisticated rules for writing forms. The NSP rules guarantee that no type error occurs concerning the interplay between the pages of the system.
  - Complex types. Arrays and user defined types may be incorporated in writing forms by using special mechanisms.
  - Native typed form parameters in the scripting language. The scripts within the server page receive the values passed via a form in the native data types of the respective scripting language, and not only as string values.
  - No unresolved links. Since NSP uses its own syntax for links and forms to other pages within the same system interface, it is possible to check, whether these links and forms point to existing targets within the system dialogue.
- Reverse engineering. From a system written in NSP the developer can generate a specification in the high level specification language Angie.

The NSP approach is oriented mainly towards building enterprise information systems [13] rather than building information architectures. NSP does not compete with content management systems or content management frameworks.

NSP is a paradigm for dynamically generated server pages that is independent from the chosen scripting language. However, for every language supported by NSP a nontrivial language mapping has to be provided. The current main development and therefore the presentation of NSP in this paper is oriented towards Java. The Java language mapping exemplifies the amalgamation of NSP with the scripting language.

The source code of an NSP page is an XML document. The language NSP is the result of changing and extending XHTML with respect to the new NSP features. A system developed with NSP runs with standard browsers as ultra-thin clients - no plug-ins are necessary.

In section 2 we motivate NSP by showing up the deficiencies of current server scripts and our theoretical model for the development of NSP. In section 3 we give a hands on introduction to NSP, afterwards an overview of the whole NSP language in section 4. Section 5 explains the use of NSP with respect to documentation and reverse engineering. Finally in section 6 we describe the relation of NSP to other approaches, which will turn out to be orthogonal to NSP.

NSP is a powerful problem oriented implementation method. It is fully integrated with a corresponding analysis method by the authors, which is called form oriented analysis [10][8][11]. This analysis method uses a conceptual model of web style applications. Such applications are called submit/response style applications. This allows further abstraction form web technology details. Form oriented analysis is strongly typed like NSP. Application models developed in form oriented analysis map perfectly to NSP technology. Moreover, the NSP dialogues derived from form oriented analysis have a clear separation of logic and presentation [9] similar, but more elaborate than Model 2 architectures.

2 Motivation

Today many web applications rely heavily on dynamically generated pages. They are often built from HTML pages with embedded server side scripts. Well known technologies in this domain are PHP, Active Server Pages (ASP), or Java Server Pages (JSP).

The interaction between browser and server is synchronous communication: the user of an HTTP dialogue causes the browser to send page requests to the server. These requests can be seen as remote method calls on the server. The HTML form standard offers a text based remote parameter passing mechanism for dynamic websites. The submitted parameters are called HTML form parameters in the following. The described parameter passing mechanism is untyped with respect to the parameter set, the single parameters, and finally with respect to the return value, namely the generated page. The same mechanism can be used in HTTP links as well: they can contain URLs following the syntax for HTML form parameter transmission. These links have to be created on the server side. NSP aims to bring the benefits of static type checking to the programming of such dialogues. For simplicity, wherever we want to reason about the system interface, we suppose that the system interface is based completely on pages generated by server scripts. Furthermore we will discuss in the following NSP in the concrete combination with Java, which also allows us to discuss the improvements to JSP achieved by NSP.

We want to recall that the JSP technology is a server side scripting technology in that it is a proposed standard technique to implement servlets [3][15][6]. JSP lets the programmer feel like programming a preprocessor for servlets, because many details of the Java Servlet API are visible.

The previously explained lack of typing in the HTTP form mechanism is not mitigated in any form by the server side scripting techniques like JSP. JSP offers the HTML form parameters to script code through an object oriented mechanism via the HTTPRequest parameter. The interesting parameters, namely the HTML form parameters provided by the forms or links calling the page, are neither typed, nor statically type checked. The static type checking is performed only on the technical level, e.g. concerning the HTTPRequest and HTTPResponse parameters. In the same way, JSP does not offer any support for creating correct calls to other JSP pages in forms and especially not in links. For links, the programmer must code the parameter passing to another server script within the result page by manually concatenating script output in the special HTML syntax.

The NSP approach does not restrict the potentials of JSP in any way, but has overcome the aforementioned problems, mainly because it is based on a clear model of page based dialogues. In the NSP model, a page has a fixed parameter list and is callable across the net. Therefore an NSP page is also named a dialogue method or method for short. The calls to this method are triggered by the user interaction with another page of the dialogue. Dialogue methods produce HTML pages as return values. The produced HTML document will be called the result page.

Each result page offers to the user the choice between different links or forms as the next step in the dialogue. These links and forms are calls to dialogue methods. Forms can be seen as editable method calls offered to the user. One of the main contributions of the NSP static type checker is to check the link and form syntax against the method declaration of the called dialogue method.

This introduces one of the most important observations in the NSP concept: from the viewpoint of the type system of NSP the generated result pages are the actual code, which has to be considered. Although the generated code is naturally not available at deployment time, NSP defines rules for writing the server script parts, the so called NSP coding rules, which are

- non prohibitive: all reasonable applications of scripting are still allowed,
- statically checkable: accordance with NSP coding rules can be checked at compile time,
- sufficient: if the NSP coding rules are followed, the question of whether the generated HTML code will be type-safe or not can be decided.

The NSP technique is independent from any particular server side scripting technology, it could even be abstracted from HTTP/HTML and could lead to a general standard for specifying page

based dialogues. Such a generalization is prepared by the notion of submit/response style applications in form oriented analysis [10].

3 A First Example

The following example points up the most important NSP notion, i.e. the type-safe call of an NSP page across the net. This minimal example allows the registration of a webshop customer.

A standard HTML page may reference the NSP page "Shop" by a hyperlink. If this link is followed by the user, an HTML page is sent that contains a link to a registration page. Note that the hyperlink to the registration dialogue method is not realized by a standard <a href> tag, but by a new NSP tag structure for links. The reason for this will become obvious later.

Within the registration page a form is offered to the user. This form sends the user input to the dialogue method "NewCustomer". An NSP form targets a dialogue method that is specified via its callee attribute. The called method has a well-defined signature. Somewhat similar to the parameter mechanism in ADA, a targeted formal parameter of a called method is explicitly referenced by its name. For this purpose the <input> tag has a param attribute. A form must provide actual parameters exactly for the formal method parameters, either by user input or hidden parameters. In the present simple example this is fulfilled. In general the overall demand for type-safe calls of NSP methods causes sophisticated rules for writing forms and a new innovative widget set.

If the dialogue method "NewCustomer" is invoked, inline Java code is executed. This Java code calls a user defined imported business method.

```
<nsp name="Shop">
  <head><title>Shop</title></head>
  <body>
    <link callee="Registration">
      <linkbody>Customer Registration</linkbody>
    </link>
  </body>
</nsp>
<nsp name="Registration">
  <head><title>Registration</title></head>
  <body>
    <form callee="NewCustomer">
      <input widget="textfield" param="customer"></input>
      <input widget="intfield" param="age"></input>
      <submit></submit>
    </form>
  </body>
</nsp>

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

4 Language Description

4.1 Top-level Structure

An NSP page consists of a head and a body. The signature of the page is defined with appropriate <param> tags between the page's head and body. The <param> tags have attributes for specifying names and types of the parameters. Java code may be placed inside <java> tags. The dialogue method parameters are accessible in the inline Java code.

In addition to <java> tags it is possible to use <javaexp> tags as a controlled variant of direct, i.e. Java coded, writing to the output stream. In NSP no special non-XML syntax for expression scriptlets like the JSP <%= %> signs is available. Because of that it is not possible 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 rather than attributes in NSP. As a result an NSP page is a valid XML 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 style sheet technologies [20].

4.2 Type System

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. In order to understand the type system of NSP one has to realize that in NSP the static type rules apply to generated forms. In NSP the following form declaration has to be considered invalid, because three input fields target the same parameter, which is not an array parameter.

```
<nsp name="M">
  <head><title>M</title></head>
  <param name="i" type="int"/>
  <body></body>
</nsp>

<!-- form inside another page -->
<form callee="M">
  <java>
  for (int j=1;j<3;j++) {
    </java><input widget="intfield" param="i"></input><java>
  }
  </java>
  <submit></submit>
</form>
```

That is, in NSP all HTML code which is created dynamically by a form declaration is not just considered as a block for capturing actual parameters. Instead it is considered as a method call offered to the user as a whole. Precisely in this sense the form has to support the signature of the called method exactly. Altogether this leads to rules for writing NSP code. These rules pay tribute to the fact that server pages are essentially a mix of HTML and scripting code. The aim of the subsection 4.2 is to elaborate these so called NSP coding rules for writing type-safe forms, which are designed in a way that the resulting notion of type correctness may be checked statically. Correct NSP code is very natural and the NSP coding rules are easy to learn. Altogether NSP contributes the adaptation of typed programming discipline to the context of programming web interfaces.

The rules for writing type-safe forms are presented as a combination of declarative characterization and informal guidelines for coding through examples and counter-examples.

4.2.1 Input Capabilities

In NSP an actual form parameter may be provided by user input or by a hidden parameter. Note that we use the term "input capability" for input fields, input controls, and - a bit sloppy, though in accordance with usual HTML terminology - for hidden parameters, too. NSP offers at least the usual HTML form input capabilities but defines their usage with respect to type safety, i.e. a textfield may be used for String values for instance, radiobuttons for enumeration types, multiple select lists for array types. But NSP improves the HTML form controls in several ways, as will be outlined now.

Consider the following problems with HTML forms. In forms often so called required fields occur. These fields must be filled out by the user. With standard HTML the developer has to deal with required input fields explicitly. For example the submission of an invalid form may be prevented with client side scripting plus disabling the submit button as described in [12]. Alternatively a server side script may check if the actual required parameter is an empty string and must provide appropriate error messages and a new input capability if necessary. A similar problem arises with the entry of integer values. Usual scripts have to react on non-convertible values with extra dialogue.

In NSP these and similar problems are tackled from the outset. Consider the following examples.

```
1 <input widget="textfield" param="s" entry="required"></input>
2 <input widget="textfield" param="s" entry="optional"></input>
3 <input widget="integerfield" param="i" entry="required"></input>
4 <input widget="intfield" param="i"></input>
```

A textfield may be specified to be required (1). The encompassing form cannot be submitted, unless this field is filled out by the user. If a textfield is optional (2) and the field is not filled out, a null object becomes the actual parameter. For textfields, the entry attribute's default value is "optional". A textfield may target only a String value. For numbers extra widgets are provided. A Java Integer object may be gathered with an integerfield (3). Like a textfield the integerfield may be required or optional. Furthermore an entered value must be a number, otherwise it is not possible to submit the form. An intfield (4) is used to target parameters of type int. It behaves like an integerfield, but it is immutably required, because there exist no canonical mapping of an undefined value into a Java primitive type. The advanced developer can customize the default behavior of the NSP widgets. Further considerations led to a new innovative set of widgets in NSP.

The NSP widget set is realized both with client side and server side scripting technology. As an example consider the realization of an Integer input field with server side scripting technology. During the parsing of the transmitted parameter stream it is checked whether the parameter is convertible. If not, the calling form is redisplayed with the already entered values as default values. The client side realizations guarantee performance.

4.2.2 Forms and Links

Note that we use the term "basic type" for every Java primitive type, for every wrapper of a Java primitive type, for String, and for every JDBC API primitive SQL type throughout the paper. For a formal method parameter of basic type, there must be exactly one input capability in the generated HTML form in the result page. The generating NSP form code must ensure this for every possible execution path.

Therefore in NSP input capability tags may occur within conditional control structures, as long as all alternative branches produce exactly one input capability for the respective parameter. Furthermore in NSP input capabilities for basic type values must not occur within loops. With these rules, the type safety with respect to basic types may be statically checked.

For example the following form declaration is considered invalid.

```
<form callee="M">
  <java>
  if (x==3) {
    </java><input widget="intfield" param="i"></input><java>
  }
  </java>
  <submit></submit>
</form>
```

Instead the following form declaration is a correct alternative.

```
<form callee="M">
  <java>
  if (x==3) {
    </java><input widget="intfield" param="i"></input><java>
  } else {
    </java><hidden param="i">815</hidden><java>
  }
  </java>
  <submit></submit>
</form>
```

Note that a static type system like NSP's cannot prevent such dynamic type errors that result from using the output stream to send dynamically generated form fragments to the browser directly. Generally, using the output stream for sending HTML 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-safe NSP system. A corrupted form that is caused by a prohibited use of the output stream may contain invalid input capabilities or a wrong number of input capabilities. This is just considered as a dynamic error, like division by zero. But even though such a dynamic type error should not occur, NSP provides sophisticated support for it. In the default case an appropriate exception is thrown. But beyond this a switch is offered by the tolerant attribute for every form. A tolerant form may have too many input capabilities for a parameter of basic type. Appropriate rules then guarantee that exactly one actual parameter is chosen. Note that it is not possible to switch off static type checking with the tolerant attribute. A form may only be tolerant with respect to provoked dynamic type errors.

In NSP the syntax for hyperlinks follows the form syntax. In NSP no tedious handling with special signs is needed in order to use hyperlinks with parameters. Links may only include hidden parameters as input capabilities. The value of a hidden parameter must be given by a Java expression.

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

4.2.3 Arrays

NSP offers sophisticated type-safe support for gathering arrays of values in forms. In the following form fragment, an Integer array parameter articleIds is targeted.

```
<form callee="N">
  <java>
  for (int i=0;i<8;i++) {
    </java>
    <input widget="integerfield" param="articleIds">
      <index>i</index>
    </input>
    <java>
  }
  </java>
  <input widget="integerfield" param="articleIds">
    <index>9</index>
  </input>
  <submit></submit>
</form>
```

All input capabilities that target the same array parameter together provide the submitted actual parameter. The index specifications must be unique. The greatest index determines the length of the array. In the example the invoked method receives an array of fixed length ten. Array elements which correspond to fields that were not filled out, are null objects. Furthermore, the ninth element is a null object, because no input capability exists for this index.

But NSP offers further features concerning arrays in forms. Consider for instance the following example.

```
<form callee="N">
  <input widget="integerfield" param="articleIds"></input>
  <input widget="integerfield" param="articleIds"></input>
  <input widget="integerfield" param="articleIds"></input>
  <submit></submit>
</form>
```

In this example the index tags are omitted. Now the invoked method just receives actually entered values. If an optional field is not filled out, no object at all will occur for it in the actual parameter array. This NSP feature frees the programmer from tedious work in many cases. All joint input capabilities of a targeted array parameter must either provide a unique index or otherwise omit the index.

In contrast to parameters of basic type no restriction on the occurrence of input capabilities is necessary, because both arrays of length zero and arrays containing null objects are considered correct.

4.2.4 User Defined Types

NSP allows the usage of user defined types in forms. One possibility to target a formal parameter of user defined type and its fields is the usage of path expressions. The following example demonstrates how a complete object tree concerning a user defined type structure may be gathered by a form this way. User defined types that should be used in NSP forms must follow the Java Beans naming convention. We use an obvious notation for user defined types in the following examples.

```
Customer {
  String name;
  Integer age;
  Address address;
}
```

```
Address {
  String street;
  int zip;
}

<nsp name="NewCustomer">
  <head><title>New Customer</title></head>
  <param name="customer" type="Customer"/>
  <body></body>
</nsp>

<nsp name="Registration">
  <head><title>Registration</title></head>
  <body>
    <form callee="NewCustomer">
      <input widget="textfield" param="customer.name></input>
      <input widget="integerfield" param="customer.age"></input>
      <input widget="textfield" param="customer.address.street"></input>
      <input widget="intfield" param="customer.address.zip"></input>
      <submit></submit>
    </form>
  </body>
</nsp>
```

Under all circumstances and for each type of a type structure, there must be either exactly one input capability for each of its fields of basic type or no input capabilities at all. Fields that have a user defined type must not have an input capability. Appropriate object references are generated automatically. Following a type structure path, the first time a type occurs that does not have any input capabilities, the generated object reference is the null object. Altogether these rules enable the usage of cyclic type structures in forms. This is exemplified in the following example.

```
A {
  A next;
  int i;
}

<nsp name="M">
  <head><title>M</title></head>
  <param name="anA" type="A"/>
  <body></body>
</nsp>

<nsp name="N">
  <head><title>M</title></head>
  <body>
    <form callee="M">
     <input widget="intfield" param="anA.i"></input>
     <input widget="intfield" param="anA.next.i"></input>
     <input widget="intfield" param="anA.next.next.i"></input>
     <submit></submit>
   </form>
  </body>
</nsp>
```

Beside path expressions, NSP offers a parenthesis mechanism for targeting parameters of user defined type, that is similar to the "with" construct of
MODULA2.

```
<form callee="NewCustomer">
  <with param="customer">
    <input widget="textfield" param="name"></input>
    <input widget="integerfield" param="age"></input>
    <with param="address">
      <input widget="textfield" param="street"></input>
      <input widget="intfield" param="zip"></input>
    </with>
  </with>
  <submit></submit>
</form>
```

Beyond the usage for abbreviation, the "with" tag becomes an important construct if a formal array parameter of user defined type is targeted, because the input capabilities of fields that constitute a single array element must be grouped together. Not all layouts may be realized with this mechanism, because it is possible that a required layout structure contradicts the necessary structure of an XML document. Therefore NSP provides another low-level opportunity to group input capabilities together via tag keys.

4.3 Functional Decomposition

NSP provides tags for the usual redirect and include directives known from the Servlet API's RequestDispatcher, but this time in the method oriented NSP style.

```
<nsp name="HelloWorld">
  <head><title>Hello World</title></head>
  <body>
    <redirect callee="M">
      <hidden param="head">"Hello"</hidden>
    </redirect>
  </body>
</nsp>

<nsp name="M">
  <head><title>M</title></head>
  <param name="head" type="String"/>
  <body>
    <call callee="N">
      <hidden param="message"> head + "World !" </hidden>
    </call>
  </body>
</nsp>

<nsp name="N">
  <head><title>N</title></head>
  <param name="message" type="String"/>
  <body>
    <javaexp>message</javaexp>
  </body>
</nsp>
```

As demonstrated in the above example, in NSP an include directive is given consequentially by a server side call of a dialogue method. Again NSP gains from its solid conceptual basis: if the Servlet API is used it must be ensured that a servlet targeted by the include directive does not send a head. This problem is tackled in NSP from the outset. The method's head is pushed into the output stream if and only if a method is called across the net.

5 Reverse Engineering of User Dialogues with Angie

NSP also aims at improving the quality of documentation of the system interface. For this purpose, NSP allows reverse engineering, namely the automatic generation of a documentation of the whole system interface in the easy to read specification language Angie. An earlier version of Angie called Gently [7] was developed by the authors as a specification language for forward engineering of system dialogues. Angie extracts from a system interface all pages with their signatures together with the contained links and forms. A further very useful information generated in Angie is the list of callers for each method. The generated Angie document is the suitable place for comments, which are passed through from the NSP files, and therefore is amenable to proposed specification techniques like [14] or [18], too. The standard reverse engineering tool of Java, namely javadoc, is not suitable for documenting servlets. Applying javadoc to the customized server script classes leads only to the documentation of the technical parameters HTTPRequest and HTTPResponse. The interesting parameters, which are significant for the business logic, namely the HTML form parameters provided by the forms or links calling the page, cannot be documented automatically.

6 Related Work

6.1 Web Services contra HTML/HTTP User Dialogues

Web Services [16] 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 can be used synchronously in a request/response style or they can be asynchronous, message oriented. Web services possess a type system, the Web service definition language [19].

At first glance, web services may seem inspired from and similar to user interfaces, using the same protocol, namely HTTP. 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, synchronous or asynchronous, 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 in our view 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. The end user can react to every input form offered to him immediately, the web service can only repeat those interaction sequences that have been in principle laid out and validated once manually beforehand. On the other hand it is clear that web service technologies are not able to address this unique feature of NSP.

6.2 Web Application Frameworks

NSP delivers genuinely new features, and is therefore not comparable to other approaches [2] [4] [5] [17]. However, NSP can be easily mistaken to be a variant of architectural approaches or approaches for separating logic and presentation. We therefore give an overview of current efforts in these two areas, which receive widespread attention. However one should note that not a single one of these projects deliver any of the unique selling points of NSP.

Server side scripting technologies, like ASP, JSP and PHP allow to embed script code into HTML, or more generally XML. None of these technologies offer NSP's static type checking. Another approach is chosen by the WebMacro [17] system, which allows the separation between Java

and HTML. It uses the Java reflection mechanism to access Java Beans from HTML Templates. WebMacro is a first example for Sun's Model 2 Architecture. Again, WebMacro does not offer a specialized support for type-safe dialogues.

WebMacro leads over to the architectural frameworks for web interfaces, which allow to create presentation layers that follow a certain software architecture. An example is Struts [5], a Framework for User Interfaces, which results in a peculiar Model 2 Architecture. Another Model 2 architectural framework is Barracuda [2]. Both do not offer the type check facilities of NSP. Another framework, which lies explicitly the emphasis on separation of content and presentation, is the Cocoon [4] framework.

6.3 The "System Calls User" Approach

In the language Mawl [1] the control flow in the server script spans the whole user session. The server script is suspended whenever a page is presented to the user. The approach can be seen as a "system calls user" approach, since the process of presenting a page to the user and retrieving the input from her has the semantics of a function call from the viewpoint of the script: the data presented to the system are the parameters of the procedure, the data entered by the user are the return values. However, Mawl allows only for one form per page, hence it abandons the core paradigm of hypertext. Principally a workaround would be possible by emulating several forms and links as one single superform. This however would lead to a bad design of systems built with this technology, which would suffer from an "ask what kind" antipattern: for every form hard-wired case structures must be used to branch the session flow. This would imply high coupling and low cohesion.

7 Conclusion

The abstract notions developed in NSP are independent from the underlying scripting language. NSP does not impose a special architecture onto the system, it does not prescribe a server side technology, it does not appear on the network protocol, and therefore needs neither client side installations, nor does NSP reduce performance. In effect, NSP causes no burden for development, openness or availability. NSP has no necessary source code footprint, no network footprint, no architecture footprint, no footprint on the running system. NSP allows the use of complex types in input forms. System interfaces developed with NSP are automatically documented by the reverse engineering tool and language Angie. The NSP technology is fully integrated with form oriented analysis, a corresponding analysis method developed by the authors.

NSP is the only approach for HTML interfaces that allows static type checking of links and forms against called server scripts and delivers parameters to server scripts type-safely and in the native types of the language. NSP makes it possible for the first time during the evolution of web interface frameworks, that the whole uppermost layer for the direct conversational interaction between browser and server is freed of type and linkage errors. To that purpose, NSP has a unique feature: it can statically check, whether the dynamically generated code will be type-safe. This is made possible by the NSP coding rules.

References

[1] Atkins, D., Ball, T., Bruns, G., Cox, K.: Mawl: a domain-specific language for form-based services. In IEEE Transactions on Software Engineering, June 1999.

[2] Barracuda. http://barracuda.enhydra.org/.

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

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

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

[6] Davidson, J.D., Coward, D.: Java Servlet Specification, v2.2. Sun Press, 1999.

[7] 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.

[8] Draheim, D., Weber, G.: An Introduction to Form Storyboarding. Technical Report B-02-06. Institute of Computer Science, Free University Berlin, March 2002. http://www.inf.fu-berlin.de/inst/pubs/tr-b-02-06.abstract.html

[9] 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, Free University Berlin, March 2002.
http://www.inf.fu-berlin.de/inst/pubs/tr-b-02-07.abstract.html

[10] Draheim, D., Weber, G.: Form Charts and Dialogue Constraints. Technical Report B-02-08. Institute of Computer Science, Free University Berlin, March 2002. http://www.inf.fu-berlin.de/inst/pubs/tr-b-02-08.abstract.html

[11] Draheim, D., Weber, G.: An Introduction to State History Diagrams. Technical Report B-02-09, Institute of Computer Science, Free University Berlin, March 2002. http://www.inf.fu-berlin.de/inst/pubs/tr-b-02-09.abstract.html

[12] HTML 4.01 Specification. W3C, 1999.

[13] Kassem, N., and the Enterprise Team.: Designing Enterprise Applications with the Java 2 Platform, Enterprise Edition. Sun Microsystems, 2000.

[14] Meyer, B. Applying "design by contract". IEEE Computer, 25(10):40-51, October 1992.

[15] Pelegri-Llopart, E., Cable, L.: Java Server Pages Specification, v.1.1. Sun Press, 1999.

[16] SOAP. http://www.w3.org/TR/SOAP/.

[17] Webmacro. http://www.webmacro.org/, 2002.

[18] Warmer, J.; and Kleppe, A.G. The Object Constraint Language. Addison-Wesley, 1999.

[19] Web Services Description Language (WSDL)
1.1. http://www.w3.org/TR/wsdl.

[20] The Extensible Stylesheet Language (XSL), http://www.w3.org/Style/XSL/.