Delft University of Technology

DETERMINING USER INTERFACE SEMANTICS USING AGENTS

By

Lein Ton

KNOWLEDGE BASED SYSTEMS

DEPARTMENT OF MEDIAMATICS

FACULTY ITS

DELFT UNIVERSITY OF TECHNOLOGY

October, 2000

Table of Contents

Abstract

Information systems are often available independent of where a user is and what time it is. Remote access to such information services, independent of time and location, is often facilitated by information providers through offering their service via the Internet. Well known examples of such services are rail travel planners.

This thesis describes an agent based approach to deal with issues that rise from the differences between user interfaces to information systems. Among such issues are the excessive time users need to spend on learning to interact with functionally similar systems with different interfaces, but also integration of related services.

The approach yields agents that interact with user interfaces to extract the model of the underlying information system. The interaction agents are capable of independently and robustly querying the information system without explicit instructions.

Acknowledgements

This report is the thesis for my masters degree in Computer Science at the Mediamatics group at the Delft University of Technology, headed by Prof. Dr. H. Koppelaar. I would like to first of all thank my coach Leon Rothkrantz for never stop being optimistic about my skills. I've appreciated working with him much and have learned that there's always a wider knowledge framework for everything. I'd also like to thank Frans Ververs for giving me the opportunity to return to the Delft University. I'd furthermore like to thank my family for never pressuring me, and my girlfriend Doris for just telling me what *not* to do. I'd also like to thank my friend Jan for his logic, Pieter for being a knowledgable peer, Caspar for being a challenging sparring partner during the last months and Edgar for accomodating and motivating me.

Introduction

Information systems are often available independent of where a user is and what time it is. Remote access to such information services, independent of time and location, is often facilitated by information providers through offering their service via the Internet. Well known examples of such services are rail travel planners. Almost all European rail companies have made their schedules available on this medium.

The web pages that give access to the rail planners all offer the same functionality, but all interfaces differ somewhat. There is no “de facto” standard for the look and feel of this kind of interface. These differences make offering information services in an alternative way through e.g. wireless devices or through a speech interface more difficult than necessary. Also creating a system where multiple similar services are integrated is hindered.

The diversity of different interfaces forces users to invest time and effort into learning how to use new interfaces, although they may already be familiar to functionally equivalent systems. Figure 1 shows how users currently use different interfaces for different services. To avoid this, several different, but equivalent information systems could be made available through the same interface.

This requires that the gap between the functional description of an interface and its information system be bridged. Figure 2 depicts a situation where an agent realizes this by finding out how an interface works and then offering the service in a standardized way.

The basic idea is that agents start an interaction with WWW-sites. This dialogue is aimed to result in knowledge about the underlying structure. Our agent is modeled after human behavior.

This thesis is about bridging that gap by using agent technology to assist with creating a unified interface to similar systems. We'll describe how an agent can be used to find out how an interface works, based on the functional specification of the interface. After an introduction to user interfaces and agent technology in chapter 1, rail planner interfaces are described and categorized in chapter 2. Then a way to specify the functionality of an information system independent of its look and feel is given in chapter 3. This chapter also introduces a way to use the specification to determine the semantics of individual interface parts. The next chapter (chapter 4) will present a framework to support this determination

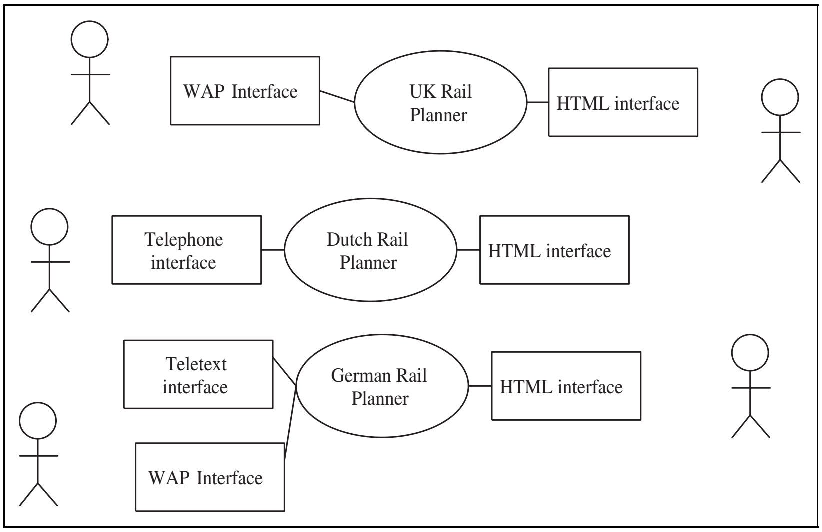


Figure 1: WWW applications with independent interfaces

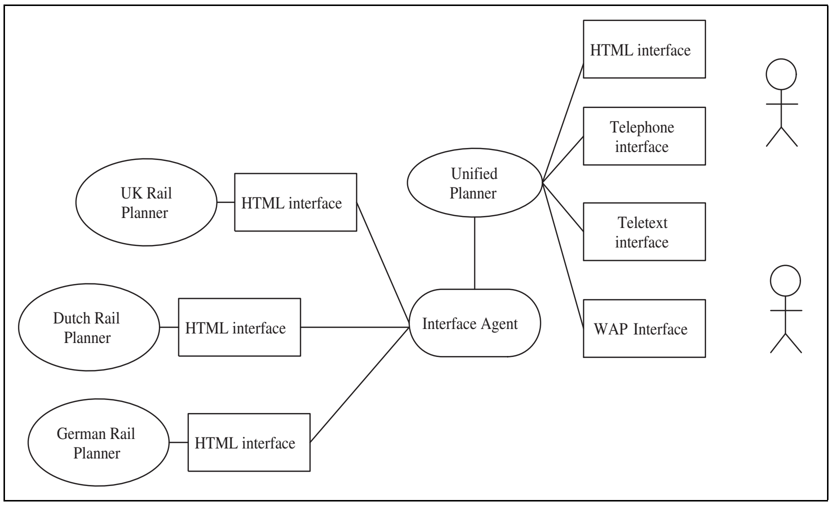


Figure 2: WWW applications with common interfaces

process. This framework contains an interface to an expert system shell that can be used to build agents that search for interface semantics. In chapter 5 two agents are shown and a test run is done. Finally chapter 6 will offer a conclusion and recommendations.

There are two appendices, the first of which contains screen dumps of seven different rail travel planners. The second appendix contains the output of a test run that was done with an agent.

Chapter 1

Background

In this chapter a short round trip through the user interface field and agent technology is given. We'll first sum up a number of agent characteristics and describe the well known software bots. These bots are involved in information retrieval on the Internet and are thus closely related to our work. Afterwards some formal methods for user interface specification are given and it is argued that we need expressibility that isn't directly found in existing formal notations.

An implementation of an interface agent that is available is described in section 1.3.

Our agents are built in Jess. An introduction to what Jess is, is given at the end of this chapter.

1.1 Intelligent agents

Intelligent agents have received a lot of attention recently. There is some confusion about what exactly agents are or when computer programs are intelligent agents. This section gives an overview of characteristics that are commonly used to characterize a computer program as an agent.

In chapter 5 two agents that extract interface semantics are described. Based on the agent characteristics described below, our agents are classified in the summary.

1.1.1 Agent characteristics

Agent characteristics have been formulated by Woolridge and Jennings [14], Nwana [11] and Maes [9].

In 1995 [14] Woolridge and Jennings gave the following characteristics for agents:
Autonomy: Agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state.

Social ability: Agents interact with other agents (and possibly humans) via some kind of agent-communication language.

Reactivity: Agents perceive their environment, (which may be the physical world, a user via a graphical user interface, a collection of other agents, the Internet, or perhaps all of these combined), and respond in a timely fashion to changes that occur in it.

Pro-activeness: agents do not simply act in response to their environment, they are able to exhibit goal-directed behavour by taking the initiative.

Later, Nwana [11] gave three characteristics that some agents need to have and others don't:
Autonomy: Taken from Woolridge and Jennings.

Cooperation: An elaboration of Woolridge and Jennings' social ability which is needed for agents to cooperate. It is possible for agents to coordinate their actions without cooperation.

Learning: A key attribute of any intelligent being is its ability to learn. The learning may also take the form of increased performance over time.

In Nwana's workt an agent is typified as either static or dynamic and either deliberate or reactive.

Maes [9] calls a computer program an agent if it is long lived, has goals, sensors and effectors and decides autonomously which actions to take. She distinguishes autonomous robots, synthetic characters, expert assistants and software agents (knowbots or softbots).

1.1.2 Bots

Probably the best known agents are search bots or robots. These kind of agents index web pages and follow links on these pages. In this way they categorize hundreds of millions of pages based on language, with hyperlinks, location etc.
The result of the work of these agents can be used through search engines like Altavista (http://www.altavista.com) and Alltheweb (http://www.alltheweb.com). With these engines all indexed pages can be searched by category or by included text. The Google search engine (http://www.google.com) additionaly orders search results by relevance. It calculates the relevance by counting the number of links a page is referenced by.

Agents are little programs that request web pages and update their knowledge about them. They can be directed to exclude certain pages or to give priority to certain pages for indexing.

Several commercial search engines make their agents available for customization, so that people can create indexes specific to their own needs. These search engines include Excite (www.excite.com), Altavista (www.altavista.com) and Thunderstone (www.thunderstone.com).

A useful toolkit for building software bots is AgentCLIPS (see http://users.aimnet.com/ yilsoft/softwares/agentclips/aginfo.html). It uses CLIPS, an expert system language with extensions that make it possible to retrieve web pages.

1.2 User interfaces

A user interface is that part of a computer system that makes the system accessible to human users. Through a user interface users can have a dialog with an information system, retrieve data or send commands.

Many information systems can be accessed in different ways. Some can be accessed as web pages on the Internet using a web browser, others can be accessed using a telephone - a suitable voice interface might have been defined in voiceML. Recently the wireless markup language (WML) has become more popular - interfaces that are suited for use on cellular phones can be written using e.g. WML. Information systems can thus have multiple interfaces that each use different means of communication.

On top of the different ways an information system can be accessed, many information systems such as library reservation systems, Internet search engines and travel planners are offered by different providers. Although these systems may offer similar functionality, they do so in a fairly different way.
A library system might e.g. facilitate a search for books on only the ISBN where others offer such a search on author and title. Different travel planners let users specify the travel month, day and year in different orders.

To make it possible to automate access to similar systems without having to use different specifications and procedures for each different interface, we need a way to describe what information an interface requires to perform its task.

Such a user interface description language will be described in chapter 3. In this section we'll offer some overview of existing techniques and related material.

1.2.1 Formal methods

With formal methods a system can be described using a precise and semantically sound language. It can be described what a system can do, rather than how it does it.

Ian MacColl and David Carrington [8] split formal methods in these three categories:

- Model based methods are suited well for specifying complex data structures. The Z notation and the Vienna Development Method (VDM) are examples.
- Property based methods define system behavior through a set of properties.
- Behavior based methods define systems in states and sequences of states.

Much of the work about applying formal methods to Human Computer Interaction (HCI) emphasizes reasoning and analysis rather than implementation. A current project at The University of Queensland [6] is concerned with formal notations for (object-oriented) specifications of user interface software.

Formal methods are also used to describe HCI. Palanque and Bastide [13] use the Interactive Cooperative Objects (ICO) formalism to specify, verify, and implement user interfaces. Yip and Robson [15] use formal methods to directly support testing user interface software. Geert de Haan describes a model to design user interfaces based on the knowledge a competent user likely has. The model is called ETAG (Extended Task-Action Grammar) [2].

Describing the human computer interaction enables testing an implementation against its specification. Also functional requirements can be checked before implementation. An example of such (behavioral) testing is verifying that the described interface can be used to search for a book when only its title is known. Another test is if certain tasks can be done within two actions.

Another reason for describing HCI is that it enables (formal) verification with e.g. temporal logic. Queries such as "is it possible to fill out the date before the time is filled out" can be answered.

1.2.2 Practical approaches

UIML Independent user interface language, UIML [1] is a meta-language, analogous to XML. Just as the XML specification does not define what tags are used, the UIML specification does not define a vocabulary of user interface components and properties. In this way UIML can adapt to various devices. Second, UIML can be translated to other languages, both imperative languages like Java and C++ as well as markup languages like HTML and WML. Third, UIML permits a description of many types of interaction of the user with the interface, without resorting to scripting or other languages. This facilitates portability of interfaces among devices.

XNS eXtensible Name Service, XNS [12] is a distributed name service where universal addresses link to data packages that may contain all sorts of personal data. Filling out XNS forms is envisioned to be done by one's personal "web agent". Such forms have fields with names that are agreed on in the XNS specification.

Great effort has been put in the HCI field of research to formalize the definition of user interfaces. This has enabled requirement testing and eased building correct interfaces. Also valuable work has been done on unifying interface description languages. A topic not well treated is the determination of functional equality of different user interfaces and the generation of functionally equal interfaces. We need this to be able to describe interfaces that we expect to encounter when we're using different interfaces to information systems with similar functionality, such as rail planners.

1.3 A combination of an agent and an interface

An interesting aid in filling out forms on web pages is the AI Roboform from Siber (see http://www.roboform.com). Many forms on HTML pages ask for the same information:

credit card number, age, name etc. This software is capable of finding out where to enter the date. Siber does not make their matching method public. They likely check names of form fields and run then against a database of known fields.

1.4 KISS

The KISS method is a set of rules of thumb that ease creating an object oriented model of an organization. The method is based on Kristen's experience that:

1. Knowledge is best specified in the jargon of the user.

2. An organization exists of a controlling and a executing part. The controlling part directs the actions of the executing part.

The KISS method (see [7], [5], [4]) offers the modeler a rich set of models of which the most important are the Subject-Communication model, the Object Interaction model and the Object Lifecycle model. The models share several abstractions such as for Subject and Action. This may ease verifying one type of model against another.

An organization is modeled and an information system is designed and implemented following the Model Cycle. This cycle consists of twenty steps including a grammatical analysis of descriptions that people who are active in the organization give of their organization. People are constantly confronted with the results of their description and asked to interact.

The process translating the interaction into the several models is intuitive but especially for text analysis directions are given as to how models and their parts are to be extracted. First all sentences are transformed into a standard format. The resulting sentences are called structure sentences. The following characteristics are a selection of those that are used to create the several models:

1. Verbs in plural form indicates an action is repeatedly done.

2. Gerund indicates actions can be ordered in time.

3. Nouns may be objects.

Interacting with a system to create a model is a method that is not necessarily restricted to modeling organizations. We will use this idea to base extracting semantics from user interfaces on.

1.5 CLIPS and Jess

Jess is an expert system shell language that has been written in Java by Ernest J. Friedman-Hill. Using an expert system a set of rules that can be repeatedly applied to a collection of facts about the world. Rules that apply are fired, or executed. Jess uses a special algorithm called Rete (see for example "Rete: A Fast Algorithm for the Many Pattern/ Many Object Pattern Match Problem", Charles L. Forgy, Artificial Intelligence 19(1982), 17-37.) to match the rules to the facts. The expert system can be defined in a language close to CLIPS.

An addition to Jess is the possibility to be seamless integrated into a Java program. This can be done in three ways:

1. Through taylor made interfaces. Additional commands can be created that take Java objects or Jess data type as an argument.

2. By reusing the Rete engine.

3. By using Jess' scripting interface.

An expert system in Jess consists of three parts:

1. Fact list: contains the data on which inferences are derived.

2. Knowledge base: contains all the rules.

3. Inference engine: controls overall execution.

A small example of an expert system in Jess is shown below. The system tries to match input fields with data that needs to be entered in one of those fields. The fields are represented as Jess assertions:

```
;; facts
(assert (input hour fieldSize 2))
(assert (input minute fieldSize 2))
(assert (input station fieldSize 15))
```

The input data is specified to be of size two as an assertion:

```
(assert (structure hour fieldSize 2))
```

A rule is defined that uses the assertions to perform the match. The rule states that if the size of the data is smaller than or equal to the size of the input field, but is not too small, there is a match:

```
;; rules
(defrule matchHour
?input <- (input ?name fieldSize ?inputSize)
(structure hour fieldSize ?structSize)
(test (<= ?structSize ?inputSize))
 (test (<= (- ?inputSize ?structSize) 10))
=>
```

```
(retract ?input)
(assert (matches hour ?name))
 (printout t "hour matches " ?name crlf)
)
```

Running the program shows that two matches are found. Following is a transcript of the run session:

```
Jess> (run)
hour matches minute
hour matches hour
2
```

Expert systems are well described [3]. In this book programming in CLIPS is treated in good detail.

Chapter 2

Rail planners

There are many services on the internet that offer rail planners. In this chapter we will analyze these planners in order to make a classification based on the similarities and differences. An internet rail planner is a web interface to a program with which train schedules can be searched. On top of this minimal service, many rail planners allow searching for the best connection between two locations or retrieving pricing information.

A rail planner interface exists of several entry forms and result pages. All services use forms to request travel data such as departure station and travel time from a user. Most services have one of those pages. We'll call these pages *start pages*. If a user forgets to specify a station, enters an invalid date or otherwise offers invalid or incomplete data, most planners present a page that shows what was wrong. Such pages usually contain a form with which the errors can be corrected. These pages are called *intermediate pages*. If all data was entered correctly, one or more schedule pages are shown. These pages hold the result of the query and are therefore called *result pages*.
In section 2.1 the characteristics of the start pages of the analyzed services are treated. Afterwards, section 2.2 deals with intermediate pages and finally the characteristics of result pages are given in section 2.3.

In order to analyze differences between rail planners, seven planners have been analyzed by having schedules made for the query "travel from A to B, depart tomorrow at 12h00". The results of this analysis are presented in this chapter.

2.1 Start Pages

A rail travel planner helps a user with creating a travel schema given a time and date, a departure station and an arrival station. Figure 2.1 shows one of the forms of the Dutch rail travel planner.

To plan a trip by train, the following information is needed:

1. Departure station

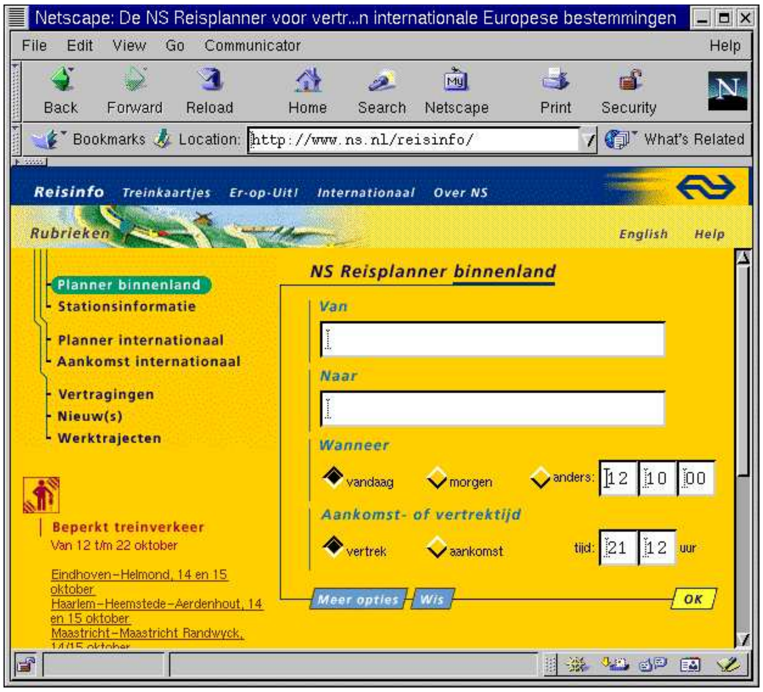


Figure 2.1: The Dutch rail travel planner

2. Arrival station
3. Arrival or departure date and time

Beside this basic functionality, some planners offer the possibility to specify after what time the trip needs to start, or the time it needs to be completed by, enter stations the train must minimally stop at or select which trains need to be used and which must be excluded. Some interfaces fill out the current time and date automatically.

Start pages contain a form that needs to be filled out and submitted in order to select and retrieve a schedule. First the look and feel of forms is described in the visual characteristics subsection, the technicalities are dealt with thereafter in the technical characteristics subsection.

2.1.1 Visual characteristics

The visual characteristics are all about how forms look, the order of fields and which fields are used for data entry. We've found the following characteristics:

- Station selection
- Time selection and time mode

- Date selection and date type
- Field order
- Departure/arrival toggle
- Pre-filled fields
- Labels and images
- Additional aides

The characteristics as they were found in the seven planners that were analyzed are described in more detail in the following paragraphs.

Station Selection We've found two ways to ask for stations - by pull down list and by free text box. The French planner offers both, the Portuguese planner first asks for the starting characters and in the next form offers all stations with these characters.

Time selection and time mode Time can be entered in a number of ways. In a text field for hour (24) and minute each; in one text field, hour and minute separated by a colon (:); in two pull down menus for hours (24) and minutes (per 15 minutes). The default Portuguese planner does not allow for a time to be specified - a user can select a time range in the extended version.

Travel planners that offer the possibility to specify if the time entered is the departure time or the arrival time usually do so by way of two radio buttons. A check box or a pull down menu would do as well. All of the examined planners asked for time in a 24 hour modus.

Date Selection and Date Type Three main ways to enter dates were found: separate fields for year, month and day, and one field for these three values separated by or a dot (.) or a dash (-). The French and Portuguese interfaces let the user select a date using pull down menus. The French uses one for day and one for month. A year cannot be specified, the months are listed as text (Janvier, .. , Decembre). The Portuguese planner asks for a year to be selected. The months are listed as numbers (1..12).

Field Order In a left to right, top to bottom sense, many of the interfaces ask for departure station first, then for arrival station. Afterwards the date is asked for and finally time can be entered.

Departure/Arrival toggle All planners except the Portuguese planner, offer a toggle with which can be selected whether the specified time is a departure or an arrival time. The Portuguese planner does not allow for a time to be specified at all.

Pre-filled Fields In almost all cases, date and time are pre-filled with the current date and time. This might give the user a hint as to where what needs to be filled out. It also makes filling out the forms in situations that occur often, such as: "Travel today and now", easier.

Labels and images All planners make use of labels: uneditable text fields or images that indicate what needs to be filled in in a nearby input field. Image labels can contain icons (little images depicting a certain meaning) and text. Recognizing such images is well beyond the scope of this thesis. Text labels contain valuable indicative information, but are language dependent and irregularly placed and will not be used.

Additional Aides The French planner offers the possibility to specify return date and time. Many planners offer a "today" toggle so that no date needs to be entered. The Dutch planner even offers a "tomorrow" toggle.

With some planners a more complex itinerary can be entered using one or more "via" fields. In some cases the means of transportation (bus, slow train etc.) can be required or excluded. The Danish and German planners offer international schedules, the former lets the user specify a country using a pull down list whereas the latter finds foreign (no German) cities automatically. The Danish planner lets you specify whether the places of departure and arrival are stations, or addresses.

The above information allows us to categorize rail planners based on the their visual aspects. Tables 2.1 and 2.2 show the results of such categorization.

2.1.2 Technical characteristics

In the first web browsers, HTML forms were rather regular and well defined. But in order to meet user demands many additions to forms have been introduced. Because of this mechanically interpreting forms has become a challenge. Different sorts of scripting extensions were added, the use of multiple submit buttons or alternatives to those buttons (e.g. images) was enabled and sessions can be used now to easily divide forms over multiple pages.

We have cornered a number of these additions to help us find out which techniques are needed to deal with HTML forms.

Sessions Interacting with an information system is sometimes done more efficiently if the interaction is split up in several stages. If e.g. an all round insurance company would sell insurances online, then they could split up questions about the desired type of insurance over multiple forms. If one form yields that the requested information is about health care insurance, it would not be useful to require a user to specify the number of wheels of the insured item on the next form.

In HTML different stages can be realized using multiple forms. But HTML is sent using the stateless HTTP protocol which means that there is no way that a web server can distinguish

Table 2.1: Rail Planner Characteristics

	Travel planner (continued in table 2.2)			
Form characteristics	NS	DB	ČSAD	Italy
station selection	text	text	text	text
time selection [1]	2 text	3 text	2 text	3 text
time type	HHMM	HH:MM	HH:MM	HHMM
time mode	24	24	24	24
date selection [1]	3 text	1 text	1 text	3 text
date type [2]	ddmmyy	dd.mm.yy	dd.mm.yyy	ddmmyyy
field order [3]	D A dmy HM	D A z t	D A z t	D A dmyHM
station field[4]	1,2	1,2	1	1
date fields[4]	1,2	1,2	1	1
time fields[4]	1,2	1	1	1,2
tomorrow selection	yes	yes	no	no
via selection	no	yes	yes	no
dept/arrival time toggle	yes	yes	yes	no
pre-filled	date, time	date, time	date, time	date, time

[1] The number indicates the amount of occurrences, text and pull down indicate a text field and a pull down menu respectively

[2] y: one year digit m: one month digit
d: one day digit

[3] D: Departure station space: next line
A: Arrival station d: Day
m: Month y: Year
H: Hour M: Minute
t: Time z: Date

[4] 1: LENGTH 2: MAXSIZE
n.a. : not applicable

Table 2.2: Rail Planner Characteristics continued

	Travel planner (continued from table 2.1)		
Form characteristics	Portugal	Denmark	France
station selection	pull down	text	text and pull down
time selection [1]	none	1 text	2 pull down
time type	n.a.	HH:MM	HHMM
time mode	n.a.	24	24
date selection [1]	3 pull down	1 text	2 pull down
date type [2]	ddmmyyyy	dd-mm-yy	ddmm
field order [3]	DA dmy	D A zt	DA dm HM
station field[4]	n.a.	1	1
date fields[4]	n.a.	1	n.a.
time fields[4]	n.a.	1	n.a.
tomorrow selection	no	no	no
via selection	no	no	yes
dept/arrival time toggle	n.a.	yes	yes
pre-filled	date	date, time	date, time

[1] The number indicates the amount of occurrences, text and pull down indicate a text field and a pull down menu respectively.

[2] y: one year digit   m: one month digit
d: one day digit

[3] D: Departure station   space: next line
A: Arrival station   d: Day
m: Month   y: Year
H: Hour   M: Minute
t: Time   z: Date

[4] 1: LENGTH   2: MAXSIZE
n.a. : not applicable

users. So what a user has filled out previously needs to be stored. This can be done in two ways (see e.g. [10]):

- Server side: saving a unique token at the client which is sent to the server after each stage. Tokens can be made using the htaccess mechanism, using cookies or by adding a hidden field to the HTML forms. This unique token can be used by the server to lookup data the user has entered before.
- Client side: saving all entered data in HTML forms as hidden fields, saving them in one or more cookies or by using HTML rewriting.

None of the travel planners we checked use sessioning.

Multiple Submit Buttons After a user has filled out a form, he can submit it by pressing the submit button. Many interfaces now offer help with filling out forms. Access to this help is sometimes offered through additional submit buttons. These buttons do not actually "submit" the form, but are used to navigate. These navigating buttons are difficult to distinguish from the real submit button. They only differ in label and name which are generally language dependent characteristics.

Submit Images Another type of submit button is a submit image. When a user clicks on such an image the form is submitted. If there are more such images or if image maps are used, it is hard to find the right button(-part) without trying out all combinations.

Image Maps There are two types of image maps: client side and server side. Areas on client side image maps are related to URLs in their HTML page and all areas can be used as different submit buttons. To find out which is the desired (real) submit button, all areas might have to be tested. Server side image maps have their functionality (submit or navigate) defined on the server. When a user clicks on such an image map, the click-location is sent to the server. The server then decides what to do. Without understanding the image, it is practically impossible to find the "submit" coordinates (the location where a user must click to submit), because all coordinates might have to be tried out. For a representative 300*50 pixel image, this would require 15,000 tries.

Submit Types A form can be submitted with the GET and the POST method. On HTTP request level, the difference lies basically in when the form content is transferred. Usually the length of form content is restricted when the GET method is used.

Client Side Scripting Two popular scripting languages have become the "de facto" standard: JavaScript and VBScript. These languages can be used to program parts of the interaction with the interface. It is generally used to check forms for completeness before they are submitted to take some load of the server. They might also be used to aid a user with filling out the fields in a form correctly. Using client side scripting can hide the exact functions of HTML fields, buttons and images. It might be only possible to check if an HTML field is used as a submit button during runtime. Although these languages make very powerful client side interaction possible, their usage makes it rather hard to interpret the meaning of a form.

Number of Forms With most of the Travel Planner interfaces it is possible to get to the desired time table by filling out just one form. Only the Portuguese planner requires two.

Table 2.3: Rail Planner Characteristics

	Travel Planner						
HTML characteristics	NS	DB	ČSAD	Italy	Portugal	Denmark	France
Sessions	form	form	form	form	form	form	form
Nr of submit buttons[1]	0	4 (6)	0 (3)	0 (3)	0 (2)	0	0
Nr of submit images[1]	1	0	0	0	0	0 (2)	0 (1)
Image map type	server					client	
Submit type	post	post	post	post	post	post	post
Scripting	none	none	Java	Java	Java	Java	Java
Nr of forms	1	1	1	1	2	1	1

[1] between brackets:nr of buttons including ones that call scripts.

2.2 Intermediate pages

An intermediate page shows up when users do not enter all necessary data or if they enter data incorrectly. Planning a rail trip using a travel planner asks a user to specify the departure station, the arrival station and the departure or arrival time and date.

Most travel planners offer support in case a station name is not entered correctly or specifically enough. The intermediate pages are all pages that a user need not encounter - they merely show because an error occurred during the specification of their trip.

The following results were retrieved by filling out just one item (either date or time or stations) on the start page if possible. If the interface would not respond, as is the case with the Italian planner for date or time when stations are left empty), invalid or empty data would be entered in necessary fields, to not accidentally retrieve a result page.

The Portuguese planner uses pull down menus for everything. This renders it impossible to get (unnecessary) intermediate pages. The French allows stations to be specified as free text. The rest of the data needs to be filled out using pull down menus as well.

Both the Dutch and the Danish planner recalculate the date 33.13.2000 rather than returning an error message. The date is recalculated 02.02.2001 by assuming the 13th month of 2000 is the January 2001 and that the 33 day of this month is the second of the next month..

The tables below (2.4 and 2.5) below show how rail planners respond to data that was typed in wrongly.

Table 2.4: Intermediate Page Characteristics

	Intermediate Pages (continued in table 2.5)			
Page characteristics	NS	DB	ČSAD	Italy
Incomplete station name	pull	pull	pull	mesg and pull
Invalid station name	pull		mesg	mesg
Empty station	mesg	both empty: no response, one empty: mesg	no resp.	popup
Empty date	28 05 00	empty	current	empty
Empty time	00 00	empty	current	empty
Valid time (12:00)	12:00		12:00	12:00
Invalid time (26:61)	26:61	26:61 + mesg	current	popup
Valid date (18.12.2000)	18.12.00		18.12.2000	18.12.2000
Invalid date (31.2.1974)	09.06.01	31.2.1974 + mesg	current	popup
Invalid date (33.13.2000)	02.02.01	33.13.2000 + mesg	current	popup

Table 2.5: Intermediate Page Characteristics (continued)

	Intermediate Pages (continued from table 2.4)		
Page characteristics	Portugal	Denmark	France
Incomplete station names	n.a.	pull	pull and mesg
Invalid station names	n.a.	mesg	mesg
Empty station	n.a.	popup	popup
Empty date	n.a.	current	n.a.
Empty time	n.a.	current	n.a.
Valid time (12:00)	n.a.	12:00	n.a.
Invalid time (26:61)	n.a.	last time that was entered correctly	n.a.
Valid date (18.12.2000)	n.a.	18-12-00	n.a.
Invalid date (31.2.1974)	n.a.	03-03-74	n.a.
Invalid date (33.13.2000)	n.a	02-02-01	n.a.

pull: pull down
n.a.: not applicable

2.3 Result pages

Time tables are represented in various ways on result pages. Certain data are always present, other information is repeated or just included once. There is also some variety in how time and date are represented.

All results are laid out in HTML tables, and all results include departure and arrival station, departure or arrival time and departure or arrival date at least once. This is where the similarities end. Since the information is usually arranged so that human reader can understand what is where rather easily, we assumed that information is grouped.

Groupings can be expressed in pieces of information being arranged relatively dependent of each other. Therefore the results below will reflect that both information being present on a result page and these pieces being ordered, characterize result pages.
The result pages were retrieved by filling out the forms with all available data (stations, time, date). If time and date were left empty, the German and Italian planner took the current time and date.

Table 2.6: Result Page Characteristics

	Result Pages (continued in table 2.7)			
Page characteristics	NS	DB	ČSAD	Italy
Number of solutions	6	3	14	10
Query is repeated (travel from .. at ..)	y	n	y	y
Format of time	hh:mm	hh:mm	hh:mm	hh:mm
Format of date	wddMyyyy	ddmmyy	ddmm	ddmmyyyy
Relative location of dept. time and arrival time	above 12	above 3	above right 14	left 10
Relative location of dept. station and arrival station	left 1, above 6	above 3	above 15	left 1
Relative location of time and date	left 1	right 3	left 2	right 1
Relative location of (dept) station and (dept) time	left 6	left 3	left 14	left 1

w: weekday (Monday), dd: 2 digit date, mm: 2 digit month, M: month (January), yyyy: four digit year, space: next line

Table 2.7: Result Page Characteristics (continued)

	Result Pages (continued from table 2.6)		
Page characteristics	Portugal	Denmark	France
Number of solutions	12	5	3
Query is repeated (travel from .. at ..)	y	y	n
Format of time	hh.mm	hh:mm	hh:mm
Format of date	dd/mm/yyyy	ddMyyy	n.a.
Relative location of dept. time and arrival time	above left 12	above 1, left 5	left 3
Relative location of dept. station and arrival station	above 12	above 1	left 1, left above 3
Relative location of time and date	none	right	n.a.
Relative location of (dept) station and (dept) time	left 12	left 1	left 3

w: weekday (Monday), dd: 2 digit date, mm: 2 digit month, M: month (in full text), yyyy: four digit year, space: next line

Chapter 3

Interface knowledge

There are many different rail travel planners available. In the previous chapter it has become clear that these travel planners have many differences, although they have very similar functionality. This is clearly seen when the Dutch and the German planner are compared (see figures 3.1 and 3.2).

As an example, a rail travel planner might have two free entry fields to fill out a departure station and an arrival station. Question is: which field is which.

When human users interact with a user interface they find out where to enter a date, how to enter a time etc. quite easily. We assume that this is because human users can recognize the layout of a page based on experience with other interfaces, and have common sense and knowledge of the information they need to enter. Users that possess the required skills and knowledge can learn how to deal with alternative interfaces to rail planners.

Experience with other interfaces may have taught a user that form fields of a certain size usually indicate that the requested data has a length that suits it. Users will not generally try to enter the name of a departure city into a field that is only two characters wide. Also fields grouped together are usually meant for related data. Many interfaces ask for time to be entered as an hour and a minute in two separate fields. These fields will often be adjacent.

One of our goals is to extract the meaning of the parts that an interface is made of, such as free entry fields, check boxes and pull down menus. To help extracting the meaning of the parts we will analyze the knowledge that is needed in this chapter.

In the next chapter it will be shown how knowledge is represented in the various parts of a framework that supports determining interface semantics. In order to use a web interface to a rail travel planner a user needs knowledge about traveling by train and skills for using web interfaces. From the previous chapter it is clear that a lot of rail planner characteristics are available that can be used to support automatic extraction of interface semantics. It is not required to know the language to deal with a travel planner, but it makes it easier.

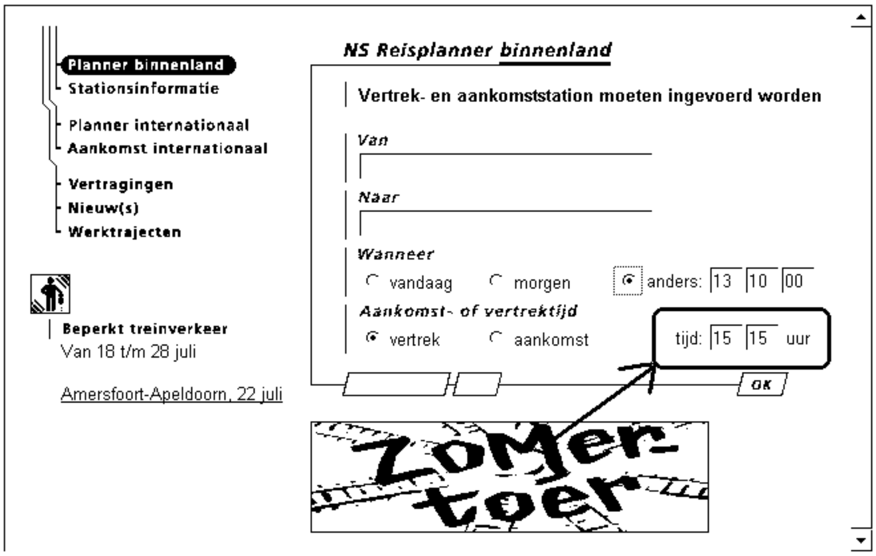


Figure 3.1: The Dutch travel planner requires that a time be entered in two separate fields.

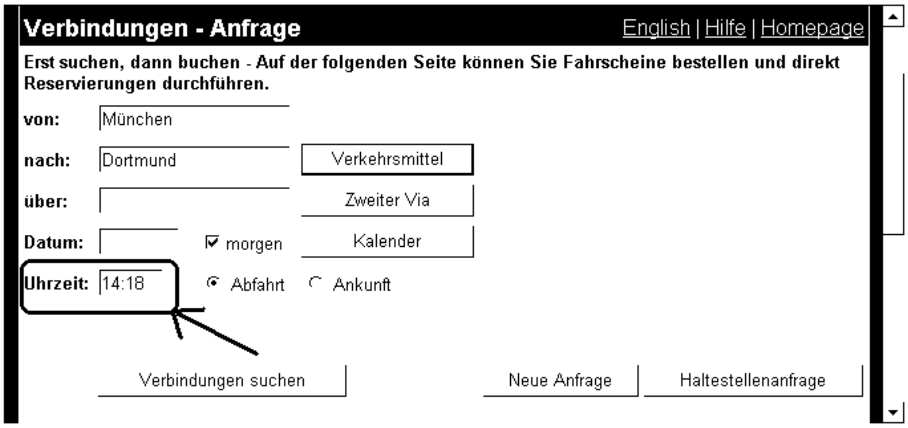


Figure 3.2: The German travel planner requires a time entered as hour:minute in one field.

The following knowledge categories can be distinguished.

- **Domain knowledge** - Knowledge that one has about rail travel planning without having used a supporting information system.
- **Logic** - Knowledge about the order to ask questions, to decide what to do in case an undesirable answer is obtained.
- **Application knowledge** - Specific knowledge about a rail planner, extras, exceptions.
- **Common sense** - Knowledge of e.g. time and geographics.
- **Skills** - Retrieve pages, understand HTML, fill out forms.
- **Interpretation of geometric objects** - “See” location, size, color, shapes.
- **Language** - Fast train in English, Schnellzug in German, Sneltrein in Dutch.

The next sections will treat each of these categories in more detail.

3.1 Domain knowledge

Information systems offer information within a certain domain. To be able to understand the information, a user must be familiar with the terminology and regulations of the domain. Also, a user must be aware what kind of information the system can offer.

In our case the domain we are modeling knowledge for is about a rail travel planner of a certain rail company of a certain country. This means that a user will expect the following basic functionality of such a travel planner:

When a user submits a query consisting of

1. Departure station
2. Arrival station
3. Date
4. Departure time or arrival time

He or she will expect to retrieve a matching travel solution. A travel solution specifies at least one combination of a departure station, a departure date and time, an arrival station and an arrival date and time.

In general a rail travel planner is able to answer the following questions:

1. Today, at what time does the train from A to B depart that arrives just before 1 p.m.?
2. At what time does the train from A that leaves for B at 1 p.m. arrive?

3.2 Logic

There are several ways to find out how an HTML interface to an information system works. One can "play" with parts of the form to see what they offer. The rules that people use for the interaction can be modeled as logic.

Examples:

1. Incorrect or incomplete data on a start page yields a directive or correction on an intermediate page.

2. A result page is shown if all data was filled out correctly.

3. Pull down menus show their options only after clicking, HTML forms may conditionally disable input fields (the date field could be disable after a user selects the "tomorrow" radio button) using extensions to HTML.

4. When checking labels, the size and default value of input fields or coloring, functions of an interface may be discovered.

5. An information system might help a user to select specific stations based on their general knowledge of cities by showing them all the train stations in a city.

The characteristics of fields and labels on an interface are an indication which purpose fields serve. Large fields will likely be meant to contain large values, check boxes are meant to choose options and labels may describe which data is expected. When searching for the meaning of individual fields, such indications can be used to reduce the possibilities. Rules (logic) could include:

1. If a field's size is larger than eight characters, it is likely meant to enter a name.

2. If two fields occur next to each other, their contents is likely to be related.

There are travel planners which require that more than one form is filled out. These interfaces are beyond the scope of our studies.

3.3 Application knowledge

We can categorize this knowledge into knowledge necessary to ask questions and knowledge needed to understand the response. The knowledge to understand responses is closely related but is directly involved in deciding how a question needs to be altered in the case of a non optimal result.

Values for Train Stations

A user needs to know values for departure station and arrival station to be able to fill these out. If he knows one value for each he might be able to find out which field on a travel planner form is for which by trying out all (two) combinations.

Users might have general knowledge but lack the specific knowledge that is required to work with an information system. As an example, a user who wants to make a train trip in the Netherlands, might know his hotel is in Amsterdam and he needs to visit a museum in The Hague. To plan a trip by train, knowing the two cities is not enough. Knowing about cities is too general; to plan a trip, a user needs to know about train stations. There are several train stations in both the Hague and Amsterdam.

Result Recognition

The interaction with a rail travel planner is meant to have rail travel schedules made. We can judge how well such a schedule was made by evaluating the response we get from the information system after sending it a query. The information system can only return a schedule if the query was formulated correctly in the first place. Three response categories can be found:

1. A schedule - the correct answer to the intended query.
2. A schedule - the answer to another query than intended.
3. A hint how the sent query is wrong.
4. An error message.
5. A new form - the information system needs more data. This response if often combined with the third response type.

Queries are formulated by filling out the form(s) that the information system's interface offers. When we receive the first type of response, we know what the individual fields on the form mean. We've reached our goal. The other responses indicate that changes need to be made to the query. The last response can also mean that the interface or the information system is unavailable.

The division into five response types was made based on recognition. In general, human users easily see with which type they are dealing. It is an interesting challenge to model this knowledge about results. We'll describe in more detail the characteristics of the different types of response below. The fifth category is not treated explicitly. The first and second type are combined and treated first.

	Halt	Datum		Zeit	Umst.	Dauer	Produkte
☑	Delft Amsterdam CS	13.07.00	ab an	8:37 9:38	1	1:01	N, Internationaler Schnellzug
☑	Delft Amsterdam CS	13.07.00	ab an	8:58 9:52	0	0:54	Schnellzug
☑	Delft Amsterdam CS	13.07.00	ab an	9:07 10:19	1	1:12	N, Schnellzug

Figure 3.3: Result of a successful query to the German rail travel planner

Van Delft naar Amsterdam Centraal
donderdag 13 juli 2000, vertrek 08:44 uur.
Netkaart: nee

Vertrek	08:28	08:37	08:58	09:07	09:28	09:37
Aankomst	9:22 am	09:38	09:52	10:19	10:22	10:38
Reistijd	0:54	1:01	0:54	1:12	0:54	1:01
Overstappen	0	1	0	1	0	1

Figure 3.4: Part 1 of the result of a successful query to the Dutch rail travel planner

Station	Tijd	Spoor	Soort	Richting/eindbestemming
Delft	V 08:28	1	Sneltrein 2218	Den Haag HS/ Amsterdam Centraal
Amsterdam Centraal	A 9:22			

Figure 3.5: Part 2 of the result of a successful query to the Dutch rail travel planner

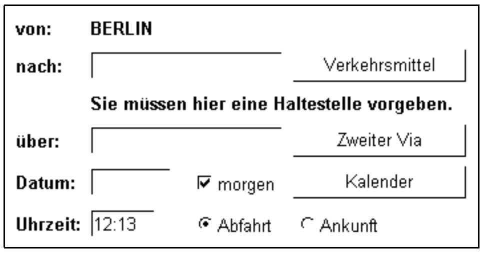


Figure 3.6: Result of an incomplete query to the German rail planner (no arrival station specified)

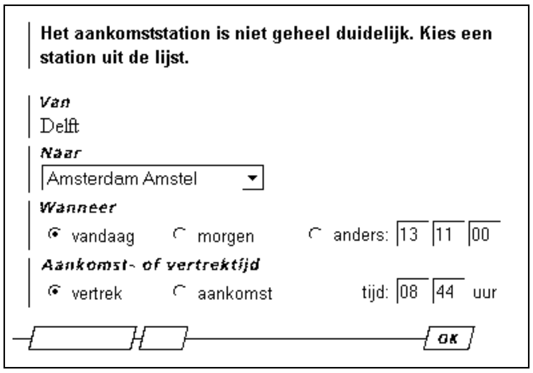


Figure 3.7: Result of a query to the Dutch rail planner with an unclear arrival station

A Schedule Figures 3.3, 3.4 and 3.5 show three results of well formulated queries. We're looking for characteristics with which we can qualify these results as correct. The two Dutch planner results are parts of one result page.

1. The first and third results contain the arrival station underneath the departure station (actually, the arrival station occurs underneath the departure station more often than vice versa.

2. The date occurs in the first and second result.

3. The departure time occurs before the arrival time in all results.

The result being a schedule does not necessarily mean that the query and the result were correct. The departure station and arrival station might have been filled out in the wrong field and characteristics as mentioned above are still needed to classify the result. More on testing such characteristics can be found in the next chapter.

Invalid Queries Figures 3.6 and 3.7 show results of an unclear question. In both queries, a departure station was given. The result page shows these again. In some cases, results contain data entered in the form if it was entered at the right place. From data occurring

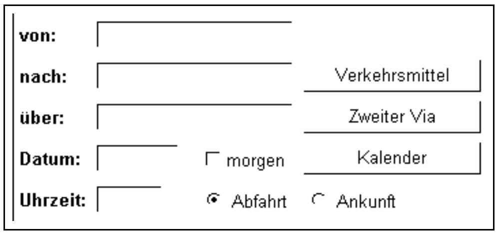


Figure 3.8: The German (Deutsche Bahn) rail travel planner

in the result page, it might be concluded which fields were filled out correctly. Based on these conclusions, there might be more information to fill out the form correctly in a next try.

Error Page Usually, error pages do not offer much support with finding out what went wrong. The information system might be unreachable, or an invalid query might have been sent. An error page characteristic is that it holds none of the query's elements.

Many characteristics can be tested by counting occurrences of keywords left of each other, above each other or occurrences in general.

3.4 Common sense

Common sense is knowledge that is not just needed in one specific situation. This type of knowledge is reusable.

Geographical Knowledge

To plan a trip with a rail travel planner one must be familiar with its location specifics. The Dutch rail planner will not respond to the Swedish city of Malmö for a departure station as it would to a valid city. A travel planner can only be used if such specifics are known.

Time and Date

To fill out a time in several different interfaces, users need to have knowledge about time. They need to know about hours, minutes, am/pm etc. The time 13:15 can be requested in different ways by the interface. Figures 3.8 and 2.1 show two different ways: The Dutch rail planner requires hour and minute entered separately, the German planner assumes a user knows how to enter a time in one field. The user interface does not initially give any hints as to how hour and minute need to be separated or if hours are to be specified in a 12 or 24 hour range.

Something similar holds for dates in these two travel planners. Besides the possibility to specify a date, the German interface offers a check box with which can be indicated that the date fields are overruled and that "tomorrow" should be taken as the date. The Dutch interface lets the user specify "today", "tomorrow" or otherwise. In the last case, the date can be filled out in three separate fields.

For filling out the time a user needs a certain notion of time - specifically minutes, hours, am and pm. Besides that, skills to apply this knowledge and perform the actual filling out are needed, since there are several ways in which time has to be specified for different user interfaces. In figure 3.1 for example, hour and minute must be put in separate fields whereas in the interface in figure 3.2 hour and minute must be put in the same field separated by a colon (:).

3.5 Skills, mechanical knowledge

Using an HTML interface requires certain skills. Such skills include filling out a text input field, viewing the options in a pull down menu and selecting a radio button.
There are basically two ways to describe those skills. Firstly, the *physical actions* one does using input devices such as a mouse or a keyboard can be recorded. Selecting the second item in a pull down menu could be recorded as: move the mouse over the desired pull down, press the left mouse button, if the second item is not visible move the mouse towards the bottom of the scrollbar at the right side of the pull down menu, press the right mouse button until the second item is visible, move the mouse towards it and click. This is a rather costly and tedious way to record skills.

Interaction with an HTML interface can be also seen as *constructing a query*, submitting it and receiving the query result. Instead of recording the actions on input devices, the query itself is the interaction atom. Usually such a query is constructed by a browser program using the data that a user filled out the form with, without the user needing to know. To construct such a query independent of user and browser, the HTML form that is normally used as a query construction aid, is first analyzed in order to find out which queries are valid. One of those valid queries is then selected. An HTML query is built up from a question mark followed by comma separated name value pairs. The pairs are separated by ampersands (&). An example query:

```
?hour=12&minute=14&departure=amsterdam
```

We've chosen to use queries as the smallest unit (atom) of interaction for simplicity reasons. Many HTML interfaces currently have non standard extensions though, that help the user construct such queries by e.g. suggesting valid dates and filling out parts of a form automatically. This sort of interaction takes place with the user locally, without intervention of the information system. Because of this decision, interfaces that build on local interaction cannot be fully dealt with using interaction where queries are the smallest unit.

Parsing HTML is a skill as well, although not many users of rail planners will directly require this skill. The parsing is generally done by a helper program (a web browser).

A user can use information that the interface gives him through labels, input field size, values in pull down menus and coloring. Some of such additional information is dependent on what a user "sees" and understands. Some users might have difficulties understanding the labeling on the German travel planner (see Figure 3.8) or might be color blind. The HTML Framework that is described in chapter 4 also extracts features. One of the things it does not see is local interaction. More on the scope of the HTML Framework can be found in the next chapter.

3.6 Interpretation of geometric objects

Utilizing shapes of buttons, the relative location of input fields, labels and images on labels requires that one can interpret these objects. As an example, the Dutch rail planner has a submit button that contains an image that consists of three parts: one part for clearing the fields of the form, one part to request a planner interface with more features and one part to submit the form (see figure 2.1). Without being able to recognize such an image it becomes practically impossible to effectively use the planner. Other geometric interpretation is concerned with recognizing which input fields are grouped. Groupings usually indicate which fields require related data, such as time and minute or departure station and arrival station.

3.7 Language

Knowing domain expressions such as station, slow train and transfer can ease interpreting both forms and results. It is not absolutely necessary to know the words though. Knowing how an interface behaves and having knowledge about representation of results will usually be sufficient. Figure 3.3 shows train types on the right side of the table. Being able to interpret these types can be combined with the knowledge that taking certain trains require a reservation to decide which itinerary is best.

Chapter 4

SMART

In the previous chapter we have described which knowledge is needed to deal with an HTML interface to an information system. Based on using rail travel planners, we could categorize this knowledge into domain knowledge, application knowledge, skills, common sense and logic.

According to Kristen [7], from interaction with an organization the semantics of this organization can be deduced. He subsequently uses the found semantics as a starting point to create an organizational model and an information system by e.g. interviewing employees and analyzing documents.

Building on this idea, in this thesis interaction with user interfaces to information systems is used to extract a model of the information system. The model is represented by labeling input fields of the interface with their meaning. A Use Case diagram depicting this can be found in figure 4.1.
Our goal is to find the meaning of the HTML fields on an interface form. We can assume that this meaning cannot always be extracted from the the form, e.g. in the case that labels and other explanatory material are given in an unknown language. To find out the meaning independent of the layout, coloring and labels of a form, we chose to interact with the interface.
Such interaction is a process guided by rules. These rules allow us to assume certain meaning in a yet unknown interface. We can then test our assumptions and change them if necessary until we find one that is correct.

The different types of knowledge are come together in **SMART**. This chapter sets apart the architecture and internal working of the parts. We'll also show how knowledge is represented in SMART.
SMART consists of an HTML Framework, a module for result evaluation, an agent and an interface between these three parts (see figure 4.2). The following sections describe these four items.

The next chapter will talk about agents. Agents contain the logic to take decisions about how to deal with an HTML interface best.

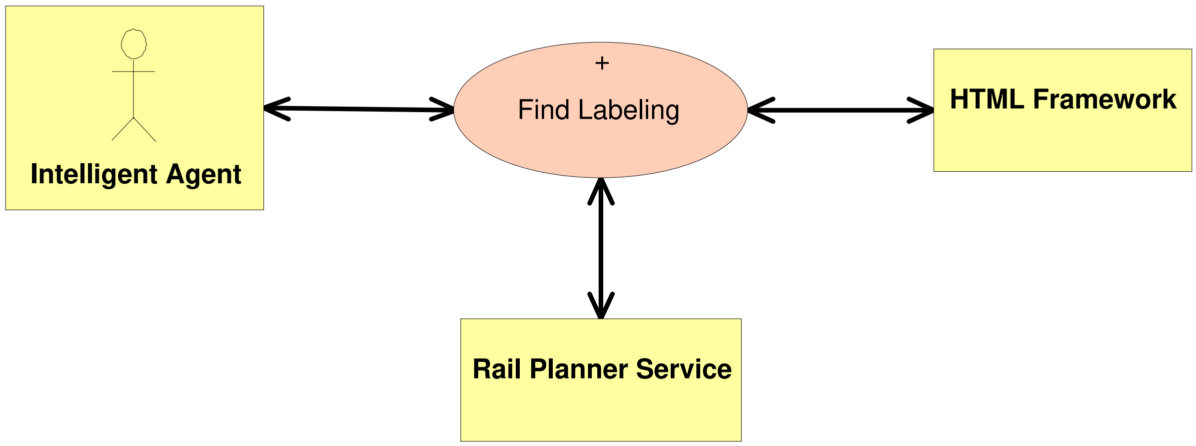


Figure 4.1: Find labeling Use Case

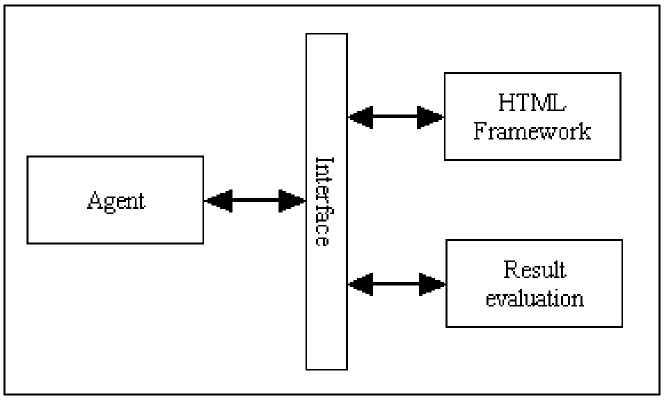


Figure 4.2: Overview of the four parts in SMART

4.1 Steps towards understanding an interface

4.1.1 Structures

Based on the examples in the last section, several sorts of knowledge have been identified. In order to enable the use of this knowledge, a representation must be made to specify it and enable it. This section introduces such a representation for start pages.

Start pages are described using *Structure*s. Structures can express either exactly which input items a user interface has or they can define which combination of Structures describe this. The former type of Structure is called Primitive, the latter is called Combined. In the following two subsections Structures are described in more detail.

Primitive Structures

With Primitive Structures we can specify which individual values have fields in a start page. Such a field could be a field for departure time, if the form has one field to enter time. It could also be a field for hour if the form has a separate field to enter a value for hour. An example Structure that specifies a choice from three options looks like this:

```
Structure("three choice") {
    type = choice;
    amount = 3;
    labels = "ch1,ch2,ch3";
}
```

Figure 4.3 shows the Kiss attribute model for this choice Structure. The Structure needs not explicitly state which HTML elements are expected on an HTML form to represent three choices.

Another example Structure (see figure 4.4) is:

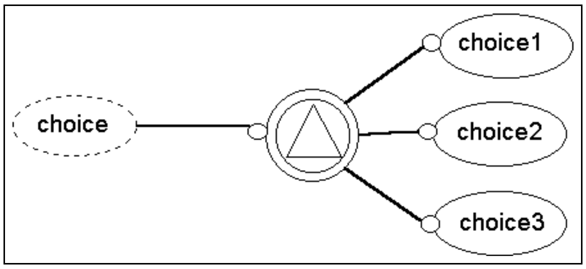


Figure 4.3: KISS attribute model for a choice from three possibilities

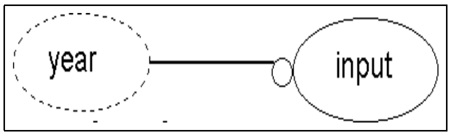


Figure 4.4: KISS attribute model for a "year"

```
Structure("year") {
    type = input;
    maxlength = 4;
    minlength = 4;
    value = 2000;
}
```

Depending on the mapping of Structures to HTML, this Structure could cover an HTML input field of length four. The HTML field this Structure is mapped on will have "2000" filled in.

Combined Structures

The two Structures for choice and year can be directly mapped onto their HTML equivalent. It is however not possible to specify all interfaces using such directly mappable Structures. Therefore Structures can be combined.

Combined Structures can be used to specify which field combinations exist on a form. An example combination is "time". To fill out the time in a form a combination of two HTML fields for hour and minute are required. Such combinations can be specified as Combined Structures using the input and choice fields. In a Combined Structure it can be stated how many and which (sub-) Structures are expected. It can also be stated which alternatives are available.

An example:
An interface can be used to enter a year. To specify knowledge that a year must sometimes be filled in as two figures (e.g. 00) and sometimes as four (e.g. 2000), Combined Structures can be used. The Combined Structure for year might look like this:

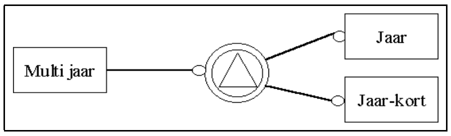


Figure 4.5: KISS attribute model for a year which can be both two figures and four.

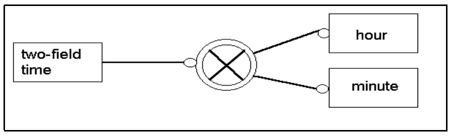


Figure 4.6: KISS attributemodel for time as hour and minute

```
Structure("multi-year") {
    type = options;
    options = ("year", "yearshort");
}
```

The HTML Framework can use this Combined Structure to search for the Primitive Structures for year and yearshort. The Primitive Structure for year was introduced in the previous paragraph as an input field with default value ”2000”. The yearshort Structure could be similar but have ”00” for a default value.
Combined Structures can be also used to define combined values. A combined value is a value that is built up from more than one value. A form that requires such a combined value has more than one input field for the value. Time in many forms is a combined value - there can be fields for both minute and hour or for one for minute:hour and a selection for am/pm. A combined Structure has an Options field, and its type attribute is set to ”options”. A Combined Structure for time could look like this:

```
Structure("two-field-time"){
    type = options;
    options = (hour,minute);
}
```

The HTML Framework can search for two suiting input fields if this Structure defines (part of) the interface.
The last two uses for ”options” can be combined. Doing this enables the definition of interfaces that require several data such as date, time, departure station and arrival station. These individual requirements may again exist of several parts. They might also define alternatives, as in the case of time (one or two fields). This way complex user interfaces can be defined.

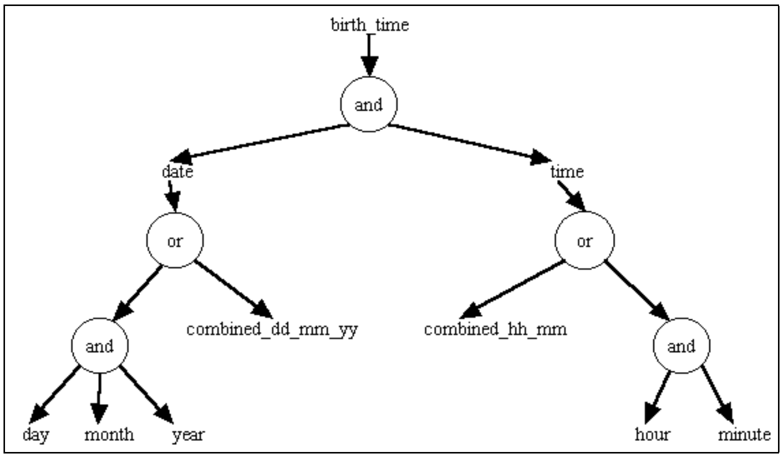


Figure 4.7: Graphical representation of rewriting a combined Structure (birth_time)

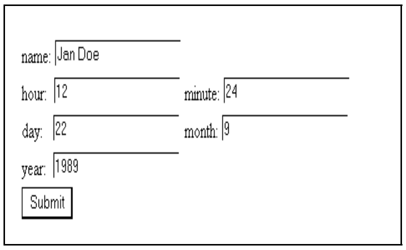


Figure 4.8: An example of a form where a name and a time of birth can be filled in

Time of Birth Interface Example

We're assuming an information system that asks for "time of birth". The following information is known about how a time of birth can be filled out with different interfaces.

1. Time of birth consists of a time and a date.
2. A date can be filled out as a day (1..31), a month (1..12) and a year (a two digit integer) in separate fields or in one field that combines day (1..31), month (1..12) and year (a four digit integer).
3. Time might also be entered in two ways: One field combining hour (0..23) and minute (0..59) or two fields, one for time and minute each.

In figure 4.7 a schema depicting the Structure that defines "time of birth" can be found. An example of an interface that matches the Structure is shown in figure 4.8.
One possible Structure that captures this knowledge is the following:

```
Structure(time_of_birth) {
    type = options;
    options = {(date,time)};
```

```
}

Structure(tijd) {
    type = options;
    options = {simpleTime, combinedTime};
}

Structure(combinedTime) {
    type = options;
    options = {(hour,minute)};
}

Structure("simpleTime") {
    type = input;
    value = "12:57";
}

Structure("hour") {
    type = input;
    value = "15";
}

Structure("minute") {
    type = input;
    value = "57";
}

Structure("date") {
    type = options;
    options = {simpleDate, (month,day)};
}

Structure("simpleDate") {
    type = input;
    value = "28.06.00";
}

Structure("year") {
    type = input;
    value = "2000";
}

Structure("month") {
    type = input;
    value = "07";
}

Structure("day") {
    type = input;
    value = "22";
}
```

4.1.2 Configurations

A Structure defines *which* input is expected, while a Configuration shows *how* this input is expected. A Configuration is a list of Primitive Structures that each represent data that is to be entered somewhere in one field on an HTML form. For example, a member of a Configuration can be a Structure that represents a time (entered in one field as 12h34).

Configurations are deduced from Structures by selecting and combining the options of the Structure.

Creating a Configuration from a Combined Structure can be seen as a non deterministic rewriting system that rewrites the Combined Structure to a string of Primitive Structures with the following rules:

1. $S_{alternatives}(x_1, x_2, .., x_n) \rightarrow x_i$ where $1 \leq i \leq n$

   $S_{alternatives}(x_1, x_2, .., x_n)$ Is short for

   ```
   Structure("alternatives") {
       type = options;
       options = {x1, x2, .., xn};
   }
   ```

   and x_i is a Combination string with one element. If x_i is a Primitive Structure, it cannot be rewritten further. A Primitive Structure is a terminal. If x_i is a combined Structure it can be rewritten using one of the two rules that are presented here.

2. $S_{combinations}((x_1, x_2, .., x_n)) \rightarrow x_1, x_2, .., x_n$

   $S_{combinations}(x_1, x_2, .., x_n)$ is short for

   ```
   Structure("combinations") {
       type = options;
       options = {(x1, x2, .., xn)};
   }
   ```

   and $x_1, x_2, .., x_n$ is a Combination string with n elements. If all elements are Primitive Structures, the string cannot be rewritten further. If the string contains Combined Structures, these Structures can be rewritten using one of the two rules.

If a start Structure is rewritten using these rules until no Combined Structures occur in the string, the string is called a Combination.

Example:
The Combined Structure

```
Structure("XXXX") {
    type = options;
    options = {xx, (yy,zz)};
}
```

is a representation of $S_{XXXX}(xx, S_{anonymous}(yy, zz)$
and is rewritten to the Configuration String
xx
or to
yy, zz

Another example:

The multi-year Structure that was introduced in the previous chapter can be rewritten to two possible configurations: `configuration(year)` and `configuration(year-short)`.
The configuration that suits the interface in figure 4.8 is: $S_{year}, S_{month}, S_{day}, S_{hour}, S_{minute}$

It is rewritten from the birth-time Structure in the following steps :
$S_{birth-time} \rightarrow$
$S_{date}, S_{time} \rightarrow$
$S_{year}, S_{month}, S_{day}, S_{time} \rightarrow$
$S_{year}, S_{month}, S_{day}, S_{hour}, S_{minute}$

4.1.3 Assumptions

After a Configuration is selected, we've explicitly stated how many fields we're expecting, what type of fields these are and what their default values are. An example Configuration for time could be: two input fields and a choice; the hour input field uses "8" as a default value, the minute field has "59" as default and the choice has two possibilities (for am and pm). The next step to take is to find a suitable HTML field on a form for each of the Configuration parts. A mapping of these parts onto the fields on the HTML form is called an *Assumption*.

The following (simplified) HTML source code of the form in figure 4.8 shows the form's HTML elements.

```
<form>
    name: <input type=text name=birthname>
    hour: <input type=text name=hour>
    minute: <input type=text name=minute>
    day: <input type=text name=day>
    month: <input type=text name=month>
    year: <input type=text name=year>
    <input type=submit value="Submit">
</form>
```

When looking for an assumption that maps the configuration S(year), S(month), S(day), S(hour), S(minute)) onto the elements of this form, one valid mapping Assumption) would be:
$S_{year} \rightarrow html(name)$
$S_{month} \rightarrow html(hour)$
$S_{day} \rightarrow html(minute)$

$S_{hour} \rightarrow html(day)$
$S_{minute} \rightarrow html(month)$

In effect an Assumption consists of mappings from Primitive Structures onto HTML elements, where the Structures that an Assumption defines mappings from are a Configuration.

4.1.4 Leftovers

An Assumption need not cover all HTML elements in a form, which means that not all elements can be filled in using the Assumption. Some interfaces may require that all elements are filled out however. The elements that are left over after filling out a form using an Assumption are called *Leftovers*. Since the interface we're expecting should be matching our starting Structure, we can assume that the elements that need to be still filled out are irrelevant for the outcome. We're therefore specifying default ways for each type of left over HTML element to be filled out. An HTML text field might e.g. be left empty, or have "NA" filled in.

The HTML Framework calculates these left over values based on a special Structure called "leftOverOptions" which defines these values. An example leftOverOptions definition is:

```
Structure(leftOverOptions) {
    type = options;
    options = {empty, choice2};
}

Structure("empty") {
    type = input;
    value = "";
}

Structure("choice2") {
    type = radio;
    amount = 2;
}
```

The leftOverOptions Structure needs to be defined so that all possible left over HTML fields can be covered, to guarantee that all fields of a form are filled out. Many interfaces don't accept incomplete submissions. In the example the assumption was chosen so that HTML(year) would be left unfilled. Structure("empty") can be used to fill this HTML field out.
Figure 4.9 shows how the HTML Framework looks.

4.1.5 Putting the terms together

The steps to find out what fields on an HTML start page mean are the following:

1. Select a start page

2. Select a start Structure

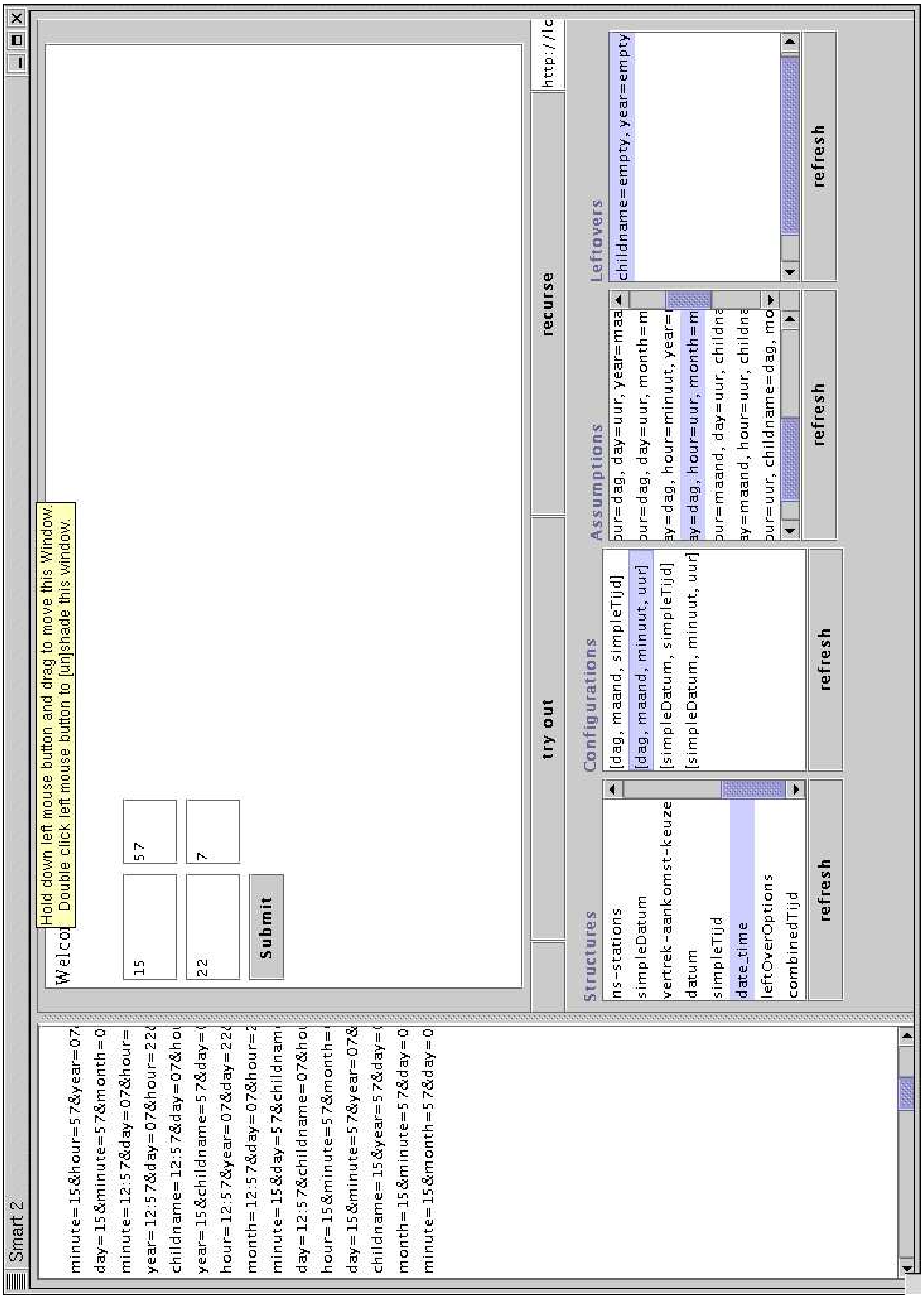


Figure 4.9: Screendump of the Framework

3. Select a Configuration
4. Select an Assumption
5. Select a LeftOver
6. Submit the start page
7. Test the result

The above list is be depicted in a Use Case Diagram in figure 4.10. This Use Case elaborates the Use Case in figure 4.1. The interaction with the Framework to find a labeling is shown in the state chart diagram in figure 4.11.

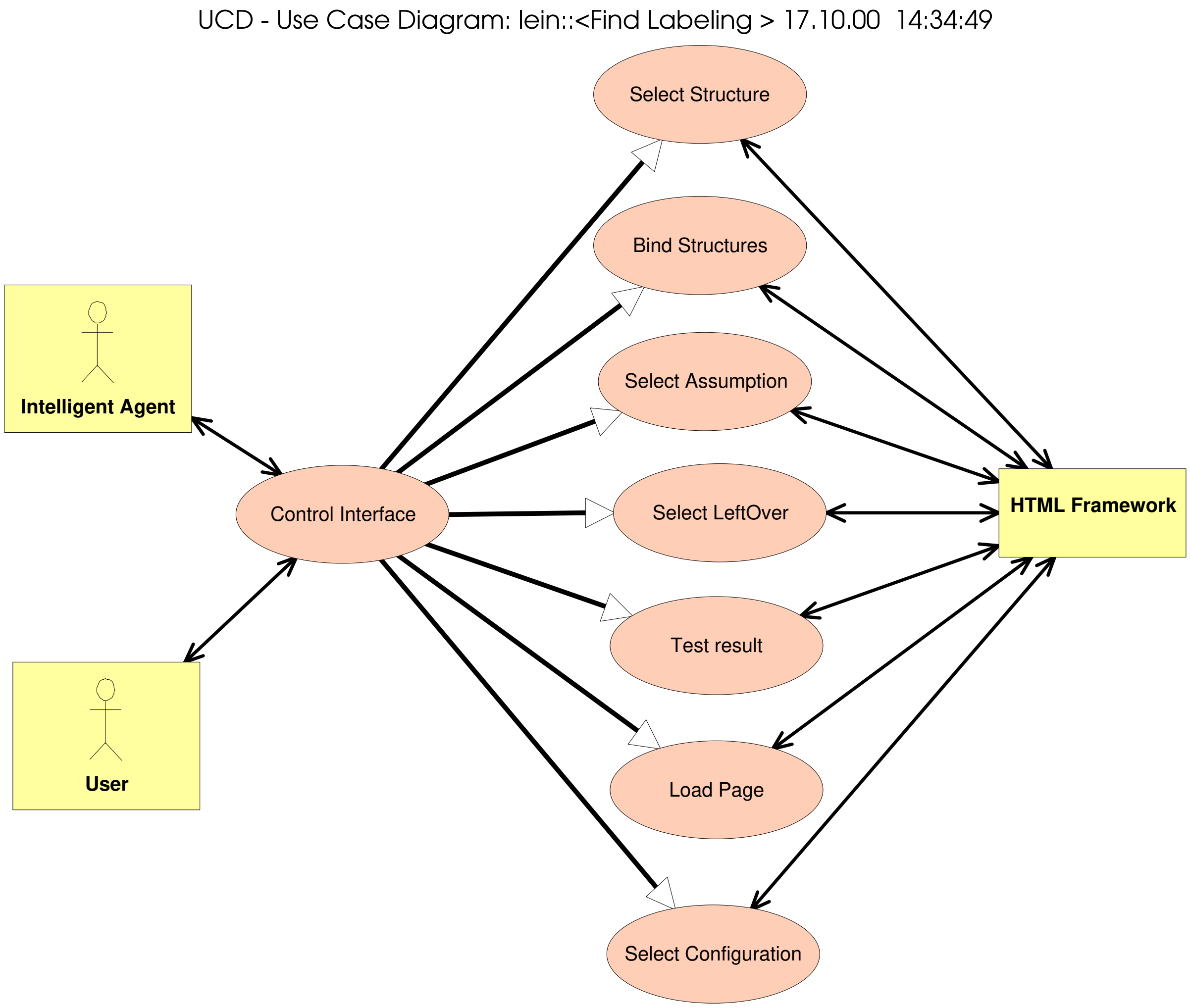


Figure 4.10: Elaborated Use Case for “Find labeling”

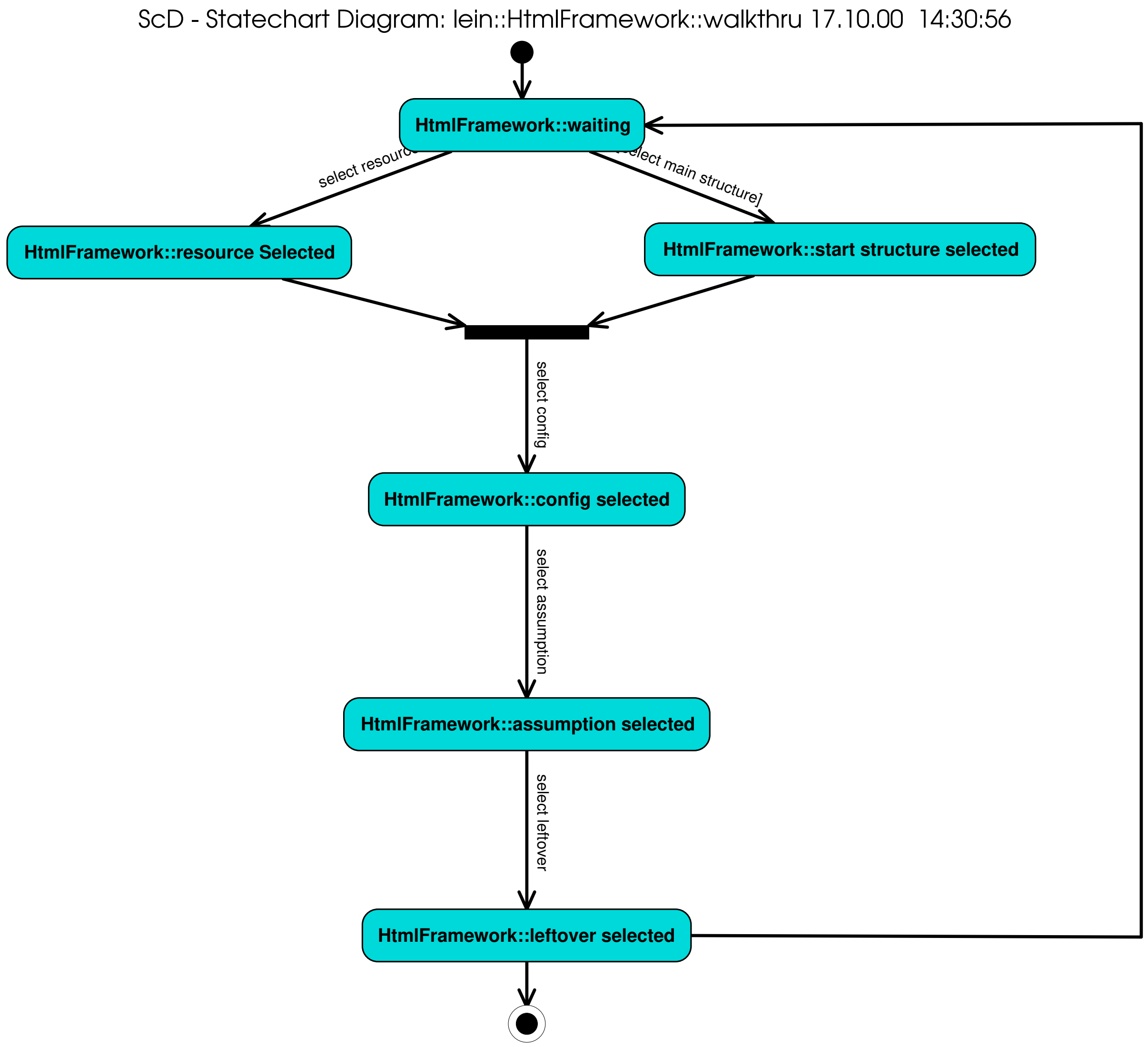


Figure 4.11: State chart of the interaction state chart with the Framework

4.2 Building blocks of the HTML Framework

The HTML Framework was built in order to:

- Send an HTML form.
- Extract forms from an HTML page.
- Read Structures.
- Rewrite Structures to Configurations.
- Combine a Configuration and a form to an Assumption.
- Combine an Assumption and a form to LeftOvers.
- Fill out a form based on a LeftOver and Assumption.

In the next subsections the HTML Framework is introduced. Afterwards the implementation of the above items is described.

4.2.1 Description

Using the HTML Framework it can be manually checked if a Combined Structure matches the interface of an information system. This is done by first selecting the desired Combined Structure from a library and selecting the starting URL of an information system HTML interface. Then a mapping from the Structure on the HTML interface is chosen in three steps. In figure 4.12 a sequence diagram of an agent interacting with the Framework is shown.

4.2.2 Extracting and submitting forms

HTML forms are forms on HTML pages. These forms are delimited by <FORM> and < /FORM> tags and consist of HTML elements in which values can be filled out and selections can be made. A form has an action property that determines where and how the form must be sent and how submitted data needs to be encoded. HTML elements that are invisible to the user can be seen as constants.

To be able to deal with HTML forms in a Java program, an `Form` class was defined with the following attributes:

- action
- elements
- hiddenVals

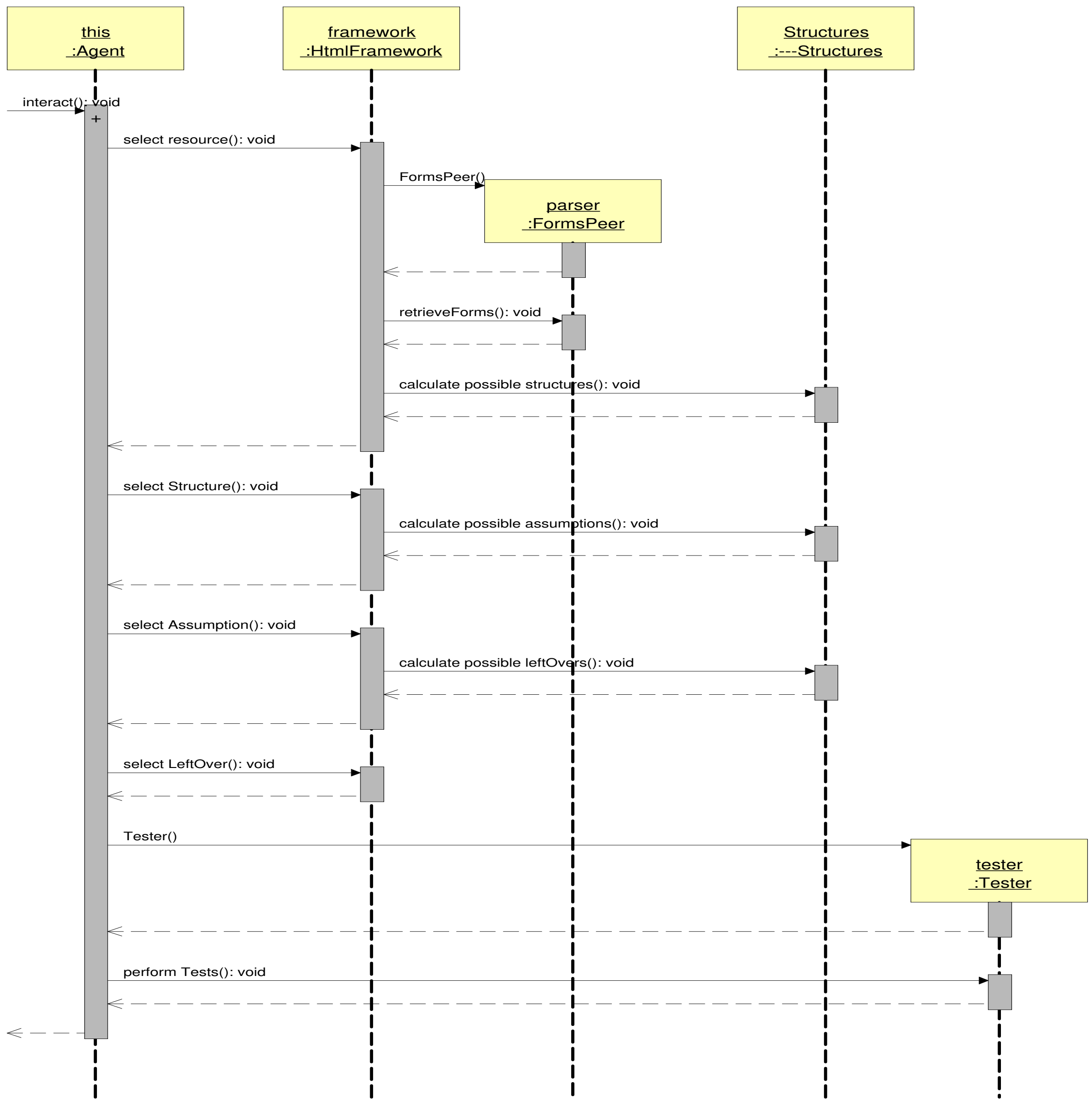


Figure 4.12: Sequence Diagram of the steps that can be done with the HTML Framework

Submitting forms results in an HTML response page from the server. This response page may e.g. contain an error message if the form was filled out incorrectly or the answer to the query the filled out form represented.
In order to keep track of which results were generated by which forms, each `Form` instance keeps track of the responses per "filling" (a combination of an `Assumption` and a `LeftOver`).

Parsing - Base Classes We're interested in the Forms on HTML pages. Our parser can thus be one that is capable of parsing HTML Forms and can be a simplified HTML Parser. The Java 2 v1.3 SDK offers a call back based HTML 3.2 Parser that can be reused by rewriting the call back methods. The Java SDK has seven classes relevant to us. They are described below. A class diagram can be found in figure 4.13.

1. javax.swing.text.html.HtmlEditorKit - To be able to deal with different kinds of content, the SDK defines an editor plug in mechanism with which support for different document Structures can be created in a universal way. The HtmlEditorKit is the plug in for the HTML Structure. Besides the HtmlEditorKit for HTML Structure, there's also an RTFEditorKit for RTF structured documents.

2. Parser, inner class of HtmlEditorKit - Actually an interface that prescribes which methods a parser that's used by the HtmlEditorKit needs to implement.

3. ParserCallback, inner class of HtmlEditorKit - In cooperation with a Reader (a class with which InputStreams can be read) this class takes care of the call backs with which an HTML document can be built up. The three most important call backs are made when a begin tag, an end tag and a simple tag are encountered.

4. javax.swing.text.html.parser.ParserDelegator, subclass of Parser - This class determines which parser is used in which cases. Normally a DTD based parser is used - the DTD can be interchanged to deal with documents with a structure other than the standard HTML 3.2.

5. javax.swing.text.html.parser.DocumentParser - The ParserDelegator uses this class to parse HTML.

6. javax.swing.text.html.HTMLDocument - An instance of this class represents an HTML document. An HTMLDocument is not very accessible. For example, it can just enumerate its HTML tags. How these tags are stored is hard to find out. that makes this class unsuitable for parsing forms.

7. HtmlReader, subclass of ParserCallback, inner class of HTMLDocument - An instance of this class recognizes all HTML 3.2 tags and sends call backs to its listeners.

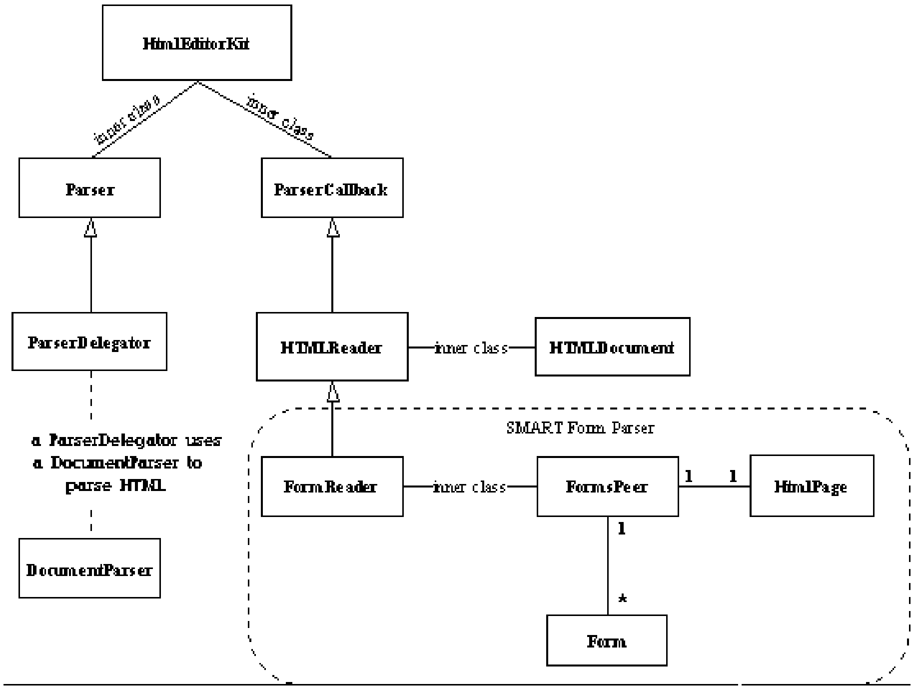


Figure 4.13: Class diagram of the Form Parser

Parsing - Forms In Figure 4.13 the class diagram of the classes described above is shown. This diagram also shows four classes that specifically take care of extracting forms (encircled with a dashed line). `FormsPeer` does the actual building of Form instances based on call backs received from `FormReader`. It also holds references to the forms it parses. Like in their parent classes, `FormReader` is `FormsPeer`'s reader. To use the Java 2 1.3 HTML Parser, a `ParserDelegator` is instantiated and a FormReader is used as its reader plug in. This `ParserDelegator` reads HTML source it receives from a `FormsPeer`.

`FormsPeer` makes use of the diffentiation `FormReader` makes between start tags, end tags and simple tags.

- FORM (begin and end)- on encountering a FORM start tag a new `Form` instance is created. The currentForm variable holds a reference to this new instance. The `Form` is added to an array in FormsPeer. A FORM end tag causes the currentForm reference to be set to null.
- RADIO - radio buttons are always grouped so that in a user interface only one radio button from a group can be selected. On encountering the first option tag of a certain group, the currentForm creates a group object (an htmlElement) and adds this object to the elements that were found previously. The option tag is added to the group object. A reference to the group object is kept in the currentElement variable. When other options of the same group are found, they are added to the currentElement.

- SELECT and OPTION - An HTML pull down box starts with "Select begin" (<SELECT>) and ends with "Select end" (</SELECT>) tags. The selectable items of a pull down box are OPTIONS. The currentForm creates a new select object (an htmlElement) on notification of a "Select begin" tag and adds all following option tags to the select object. CurrentForm keeps a reference to the select object in currentElement. On notification of a "Select end" tag currentElement is set to null.
- HIDDEN en IMAGE - a hidden tag or an image tag aren't being added to the form elements - they are used to generate a piece of an HTML query (like ?hiddenName=John) and are concatenated to the hiddenFields String variable of currentForm. The coordinates of IMAGE queries are set to (0,0). This simulates the behavior of a user clicking in the upper left corner of the IMAGE in the HTML interface. Some information systems don't work without this trick (see also section Technical Characteristics).

An advantage of this kind of parsing with call backs is that HTML Forms that contain errors can be read.

4.2.3 Implementation of representations

Implementation of Structures and Configurations

Structures hold knowledge of how forms can be filled out. For example it can be specified that a form expects a time and a date, that time would have to be specified in three fields or just one and that forms expect time in one or two fields. To be able to identify a Structure, it needs a name. A Structure is represented by the `Structure` class, its name is held in a String. The other properties of a Structure, such as minimal length, default value and type, are saved in a `Hashtable`.
The initialization of Structures is taken care of by the `SlotEnumerator` class. Calling its static getStructHash() returns a `Hashtable` containing all Structures.

There are primitive and combined Structures. A primitive Structure can be mapped one to one onto an HTML element. A combined Structure can be rewritten as on or more combinations of primitive Structures.
A configuration is obtained by rewriting a combined Structure until it just holds primitive Structures. This is done by the static method `SlotEnumerator.getOptions()`. A `Configuration` is a `Vector` of primitive Structures. The `getOptions()` method returns a `Vector` containing `Configuration` `Vector`s. The method is recursive and identifies two cases:

- The Structure of which the Configurations need to be calculated is primitive - In this case, a Vector with one Configuration (Vector) containing the primitive Structure is returned.
- The Structure is combined - the Configurations of each option are calculated using getOptions (recursively). These Configurations are then combined to one Configuration. This combination can be divided in two cases:

- config1 = [[a,b], [c,d]] and config2 = [[e,f], [g,h]]. This yields: [[a,b,e,f], [a,b,g,h], [c,d,e,f], [c,d,g,h]].
- config1 = [[a,b], [c,d]], config2 = [[e,f], [g,h]] en config3 = [[i,j], [k,l]]. Here the first two Configuration results are combined like at the first item. This results in a Configuration that can be combined with the third Configuration.

Implementation of Assumptions

The HTML Framework calculates all possible mappings using the `Tester` class. This class has rules to decide which primitive Structure in a Configuration suits which type of HTML field.

The following HTML elements are checked for suitability:

1. <SELECT> .. </SELECT>
2. <INPUT TYPE=TEXT>
3. <INPUT TYPE=RADIO>

These are the rules that are applied by the `Tester` class:

1. A primitive Structure of type input matches all <SELECT> .. </SELECT> HTML elements that contain the (default) value of the input Structure.
2. A Structure of type input matches a <INPUT TYPE=TEXT> HTML element if the length of the Structure's (default) value is lesser or equal than the length of the HTML element.
3. A radio Structure matches a <INPUT TYPE=RADIO> HTML element if the number of choices in the Structure is smaller than or equal to the number of choices of the HTML element.

An Assumption is a mapping of the primitive Structures of a Configuration onto the HTML elements of a form.

An example:
A form has three input fields (inp1, inp2 and inp3) and two radio buttons in a group. The example Assumption has two primitive Structures of type input. A possible mapping would be: prim1 $\rightarrow$ inp1, prim2 $\rightarrow$ inp3.

Both primitive Structure and HTML elements have properties such as type, size, minimal length. A method `Tester.elementSuitsSlot` was implemented that judges if an HTML element can be used to map a Structure on.

An assumption is made by assigning HTML elements in a Form to all primitive Structures in a Configuration. The assumptions are made by calling the `Tester.setCandidates()` method. This method works as described below.

Suiting HTML elements are selected from the HTML elements of a Form for the n^{th} primitive Structure of an Assumption. All preceding Structures have already gotten an HTML element assigned. For each combination of the n^{th} Structure with an HTML element an assumption is created. For Structures that have more options (radio buttons) an Assumption is created for each option. These assumptions are gathered in a result Vector. The `setCandidates()` method is now recursively called for the $n+1^{th}$ Structure. The two result Vectors are then combined.

Not for all primitive Structures a mapping onto an HTML element can always be found. Such Structures are skipped. This fact could be used by an Agent or human user to conclude that the selected Configuration is incorrect or that the information system has more than one form. This data has not been made available directly to agents or human users but can be deduced from the form submission result.

Implementation of LeftOvers

If all primitive Structures have been assigned an HTML element, there might be elements left unfilled. Some interfaces require that such fields are filled. The fields that were left over are filled using the "leftOverOptions" Structure. This combined Structure defines a mapping for primitive Structures onto all possible HTML fields, guaranteeing that all fields can be filled out using this Structure.

Leftovers are identical to an Assumption. It is a mapping from primitive Structures onto HTML fields. To determine the different possible ways to fill out the leftover fields, different LeftOvers are created by assigning a suiting Structure to each left over HTML element. If more Structures suit or if an HTML element can be filled out in multiple ways (like a pull down menu or radio buttons), all suiting combinations are made. The combinations are stored as Assumptions - All Assumptions are combined to a result Vector of Assumptions.

The class `LeftOverQuery` matches all individual HTML elements with the options of the leftOverOptions Structure using the same rules as the `Tester` class described above.

The class diagram in figure 4.14 shows an overview of the HTML Framework. Figure 4.15 shows the visual part of the Framework. Using the various selectors, Structures, Configurations, Assumptions and LeftOvers can be selected and HTML pages can be viewed.

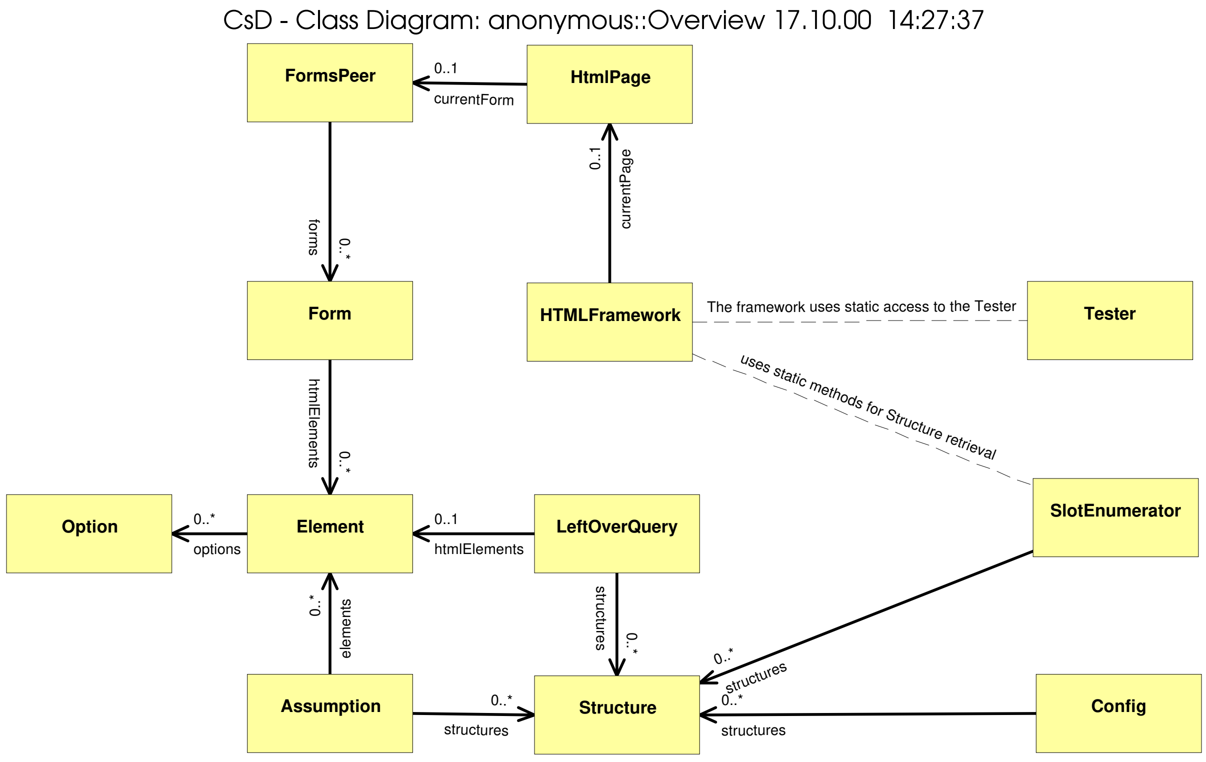


Figure 4.14: Class diagram of the Framework

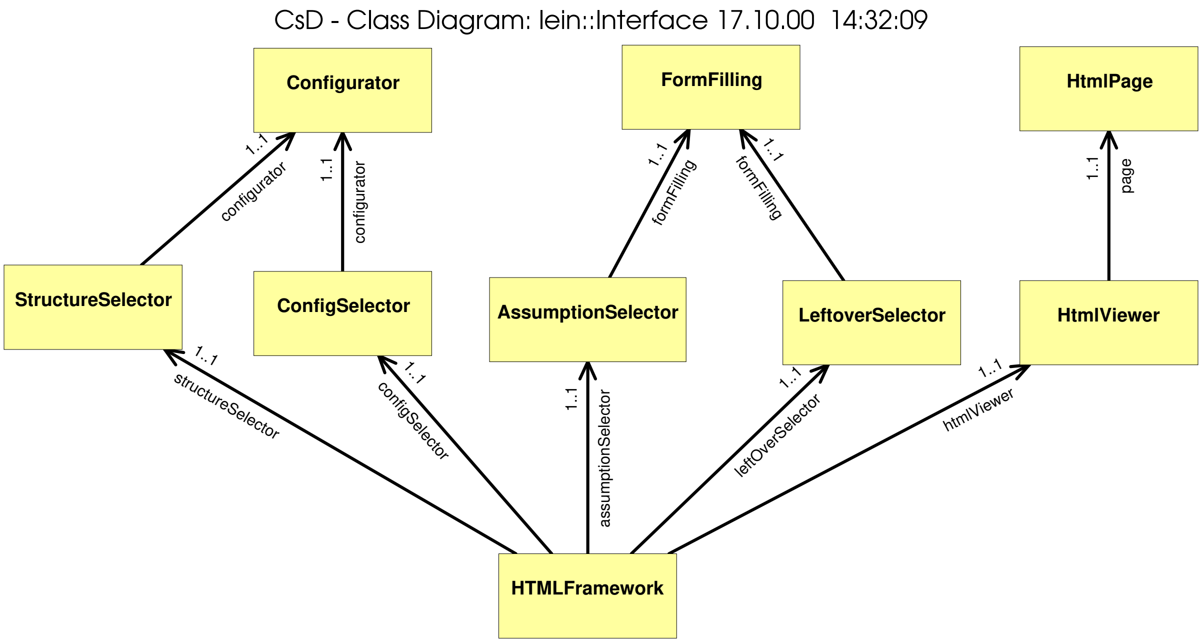


Figure 4.15: Class diagram of the Framework's interface for selecting representations

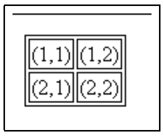


Figure 4.16: Simple table

4.3 Result evaluation

To find out if a form was filled out correctly, after submitting it, the submit result needs to be evaluated. In chapter 3 result pages were categorized in successful results, unsuccessful results and error pages. This categorization was made based on how well the Structures used to fill out the form are represented.
The result evaluation module introduced in this section assumes that results are presented in tables. It also assumes that checks for existence of a Structure representation and relative location of such representations are sufficient.
In the next subsection the way HTML tables are treated is shown. After that the available tests are described.

4.3.1 Tabellarization

All of the evaluated results that weren't errors were organized in tables. HTML code for a basic table looks like this:

```
<TABLE>
    <TR>
        <TD>(1,1)
        </TD>
        <TD>(1,2)
        </TD>
    </TR>
    <TR>
        <TD>(2,1)
        </TD>
        <TD>(2,2)
        </TD>
    </TR>
</TABLE>
```

In Figure 4.16 example output of this table is shown.
Three observations led to the design of a table parsing algorithm:

- Result pages may contain multiple tables.
- Tables may be nested.
- Tables do not exclusively use the <TR> tag as a table row separator, but also use
 within table cells.

The evaluated result pages showed it is necessary to take into account all levels of nesting in a particular table. This means that if a certain table contains several others, then containing table cannot be ignored. The Table parser is built very much like the HTML parser described in the beginning of this chapter. It also is based on the HtmlEditorKit (see subsection 4.2.2) and uses a call back mechanism.

Individual tables are stored as class Table instances. The Table class can be used for left to right, top to bottom table construction and supports the
 as a row separator within table cells.

The HtmlEditorKit's call back mechanism is taken care of by TableReader, a subclass of HTMLDocument.HTMLReader. This class supports nested tables with a stack mechanism that temporarily stores tables. If e.g. Table A contains table B, when parsing B, table A is put on the stack. Parsing such tables A and B would result in two tables as follows:

<table><tr><td><table><tr><td>W</td><td>X</td></tr><tr><td>Y</td><td>Z</td></tr></table></td><td>B</td></tr><tr><td>C</td><td>D</td></tr></table>

would become

	B
C	D

and

W	X
Y	Z

.

4.3.2 Standard tests

Result pages have all shown to contain result representations of Structures. It turned out though, that mere containment does not guarantee that the query that produced the result had a correct mapping for the contained Structure representation.

For example, if the default value for the month Structure would be "12" and it would be used to fill in the HTML element for minute, the result page may yield "12". This would have been caused by the information system that recognized "12" as a valid minute and represented it on the result page.

Many such wrong mappings can be filtered by checking where result values are positioned. If the value for minute turns up left of the value for day, this would show a wrong query in several interfaces. Also a result with departure station occurring above arrival station more often than vice versa could indicate that the right HTML fields for arrival station and departure station were chosen in the form.

For these reasons, three tests were defined:

- Existence: tests if a value exists in any of the tables on a result page.
- LeftOf: Tests whether a value was found on the left side of another in any table on the result page.
- Above: Tests how many times a value occurs no more than n rows above another. Tests are done per table, all tables are tested.

To ease searching for values that might have several appearances, all values may be specified as regular expressions. This regular expression functionality is taken from gnu.regexp (see http://www.cacas.org/java/gnu/regexp/credits.html).

4.4 Interfaces to the HTML Framework and to result evaluation

In the previous two sections, result evaluation and a Framework to deal with HTML, forms, and mapping of Structures onto those forms were described. In order to combine this functionality and to enable an intelligent agent to perform the Structure testing and evaluation, an interface is defined. The interface specifies how such an agent and those modules are glued together.

Our agents are written in the Jess language. Jess is an expert system shell written in Java. More on intelligent agents can be found in the next chapter. More on Jess can be found in chapter 1.

The Jess language can be extended with user defined functions. These functions are written in Java and implement the jess.UserFunction interface. Two such extensions were written to implement the interface: `ResultEvaluatorConnect` and `PlaygroundConnect`. They are described in the next two paragraphs. The Jess language and Java can exchange values - Jess can handle Java instances and Java can deal with Jess' values.

4.4.1 Connection to the HTML Framework

`PlaygroundConnect` adds the following commands. All commands refer to the HTML Framework which was described in the beginning of this chapter.

- `struct(int n)` Selects the n^{th} Structure in the HTML Framework through its structBrowser.
- `config(int n)` Selects the n^{th} configuration in the HTML Framework by selecting the n^{th} configuration in the configBrowser.
- `assumpt` Selects the n^{th} assumption in the HTML Framework by selecting the n^{th} assumption in the assumptionBrowser.
- `leftovers` Selects the n^{th} leftover in the HTML Framework by selecting the n^{th} Structure in the structBrowser.
- `structHash` Retrieves a Hashtable with all active Structures.
- `configVector` Retrieves a Vector with the Configurations currently displayed by the HTML Framework.

- `assumptVector` Retrieves a Vector with all Assumptions currently selectable in the HTML Framework.
- `leftOverVector` Retrieves a Vector containing all leftOvers that are currently listed.
- `pushDown` Pushes down wrong assumptions. Described in detail in chapter 5.
- `configPushUp` Pushes up Configurations that are more likely to suit an interface than others. Described in detail in chapter 5.

4.4.2 Connection to the result evaluation module

`ResultEvaluatorConnect` adds the commands:

- `leftOf` returns the number of times the first regular expression occurs left the second on the tested page.
- `contains` returns true if the tested page contains the specified regular expression.
- `setCurrentPage` sets the current HTML page in the Framework.
- `loadPlaygroundPage` retrieves the current HTML page from the Framework.

The commands listed above give Jess programs access to the HTML Framework and the Testing Module, so that their use can be effectively automated. The agents that are introduced in the next chapter are programmed in Jess and can thus perform extraction of semantics supported by the Framework and the Testing Module fully.

Chapter 5

Rail planner agents

In chapters 2 and 3 rail planners were analyzed and knowledge needed to use interfaces such as rail planners was classified. In chapter 4 a Framework and an interface description language to interact with HTML based interfaces were introduced.

In this chapter interaction with an HTML interface will be used as a means of determining which HTML fields represent what. Agents are used because they have the advantage that they can do the interaction automatically. The work in the previous chapters is used to build agents that are able to interact with HTML interfaces. In this chapter two such agents are presented. The first agent searches for the interface semantics using brute force search, the other truncates the search space by "pushing away" unlikely meanings and is called the push agent.

The agents are programmed in Jess. Jess is an expert system shell that combines facts and rules in a forward chained way.

5.1 Tasks of an HTML interaction agent

Three different tasks can be distinguished when an agent interacts with an HTML interface:

1. Determine which is the correct Configuration.
2. Determine which is the correct Assumption for that Configuration.
3. Determine which is the correct LeftOver for that Assumption.

Correct and Incorrect Configurations A Configuration is correct if all its primitive Structures can be correctly mapped onto an HTML element (such a mapping is part of an Assumption). A Configuration is incorrect if no such mapping exists.

Determining whether an Assumption/LeftOver Combination is correct can be determined by testing the HTML result page. A result page is the result of submitting the HTML form filled out based on the Assumption and the LeftOver.

5.2 Testing start pages

To find out what HTML elements mean, we need to find the suiting Structure. In the easiest case there is a one to one mapping: for each HTML element a Structure can be found and vice versa.

If there are too many HTML fields, there might be two reasons for it. Firstly unexpected functionality such as a "via station" could be present. Secondly it may mean that a wrong Configuration was chosen - for example one that assumes that time requires one field, where the interface requires two. If there are too many HTML fields, and it is assumed that the Configuration was chosen right, all Structures must be associated with a suitable HTML element. Whether the Configuration is correct or not can become clear after testing its Assumptions.

If there are too few HTML elements, a wrong Configuration might have been chosen. If it is assumed that the Configuration was correct, the information system interface might use multiple forms (see the next paragraph).

There a two types of forms:

1. **Single forms**: If the form is filled out correctly, the result page will show the desired information.
2. **Multiple forms**: The input fields are divided over multiple forms. These forms need to all be filled out correctly to yield the desired result.

Most rail planners that can be queried using just one form offer multiple forms to assist users who filled out their form partially or partially correct. An interaction tree can be made from an interface with multiple forms.

5.3 Testing result pages

Result pages can be tested with the ResultEvaluator in chapter 4. With this class it can be determined which elements (Strings) occur on the page and which relative position they have. Based on this information, an agent must decide what to do next. An agent needs to be capable of determining when it's ready. It must have a *stop condition* expressed in tests of a result page.

Occurrence of Values on a Result Page Some tests can be directly related to primitive Structures. If a primitive Structure has a default value, this value can be searched for on the result page. The occurrence of the value does however not guarantee that the Assumption was right (the HTML field that the primitive Structure was mapped on was the right one). The default value of a primitive Structure for a day could be "24" and this value would make a valid minute as well. Also the Interface need not necessarily return a page containing the value. Tests that can be related to primitive Structures are called *primitive tests.*

Absence of Values on a Result Page Absence of a value on a result page need not always mean that the Assumption (mapping of a primitive Structure onto an HTML field) was incorrect. An example of where filling out a right value does not result in that value occurring on the result page can be found at the Dutch rail planner. There are individual input fields for day, month and year and there are three radio buttons to select if the time table should be today's, that of tomorrow or another date. If this third radio button isn't selected, the date entries are ignored. Another case in which absence isn't caused by a wrong Assumption is if with the Dutch rail planner, the month is filled out correctly, but the year and the day have swapped places. The rail planner responds to such an error (filling out 00 07 05 for dd mm yy) by returning a result page containing the last date the planner is valid (09 06 01). The month was filled out correctly, but did not occur in the result page because the date as a whole was filled out wrongly.

Passing primitive tests does therefore not always guarantee that a primitive Structure and an HTML input field match. Tests that go further in guaranteeing this use the relative location of results. These tests are called *indirect tests*. An example of such a test is a test for time: if the hour occurs left of the minute or if the hour and the minute occur separated by a colon (:), time was filled out correctly.

5.4 The brute force agent

The first agent we're talking about in this chapter is the Brute Force agent. This agent will check all possible ways to fill out an HTML form based on a Structure. The steps this agent takes are:

1. Select a start page.
2. Select all Configurations one at a time.
3. For each Configuration, select all Assumptions one at a time.
4. For each Assumption, select all LeftOvers, one at a time.
5. Submit all Configuration/Assumption/LeftOver combinations and test the result until a stop condition is reached.

An agent that works this way needs to check all possibilities to fill out a form. For a real life information system such as a rail planner this is not very efficient. An advantage though is that if there is a solution (correct Configuration/Assumption/LeftOver combination), and the tests are defined well, the solution will be found. To give an idea about the number of combinations that need to be verified: The birth-time Structure (see chapter 4) can be used to search for a date and a time. Four Configurations exist for this Structure:

1. SimpleDate, simpleTime

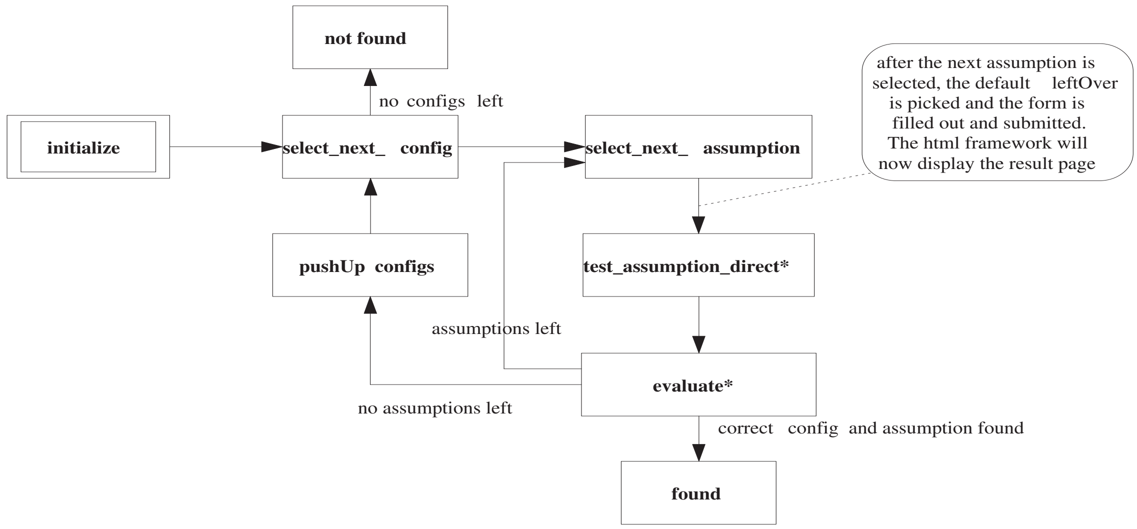


Figure 5.1: Flow diagram of the push agent

2. SimpleDate, minute, hour
3. Day, month, year, simpleTime
4. Day, month, year, minute, hour

The first Configuration has two Assumptions, the second has 60, the third 270 and the fourth 2520 ($\frac{7!}{2!}$), all Assumptions have six leftovers, so in total there are 2*6 + 60*6 + 270*6 + a second, it will take the agent up to about five hours to check all combinations.

5.5 The push agent

This agent starts checking all possibilities, but tosses away Assumptions that are unlikely to be correct (Assumption PushDown) and sorts Configurations so that those that match best are tested first (Configuration PushUp, see figure 5.1).

Initialization In the initialization phase, the HTML Framework is set up, the Structures are being initialized, a start page is selected and some variables are set (such as current_config=0). The Structure that describes the expected interface is chosen and its Configurations are calculated by the Framework.

Selecting the next Configuration The current Configuration (pointed at by current_config) is selected by the Framework and its assumptions for the current page are calculated. After all Configurations are dealt with on entering this phase, the agent will enter the "not_found" state. In that case no correct solution was found.

Selecting the next Assumption The next assumption is selected by the Framework, its leftovers for the current page are calculated and the first LeftOver is selected. The Framework then calculates the query and submits it and loads the result page.

Direct Tests In this phase, the "occurrence tests" are done. These are tests that check if certain values (Strings) occur in the result page. The values are similar to the default values of the primitive Structures in the selected Configuration. The agent assumes that if such a value cannot be found, it was filled out in the wrong place. This means that the Assumption mapped the primitive Structure onto the wrong HTML element. If the value was found a primitive assertion is done that can be used in the next phase to determine which mappings in the Assumption were correct.
The occurrence tests that did not find find values for a certain primitive Structure request an Assumption pushDown from the Framework. This filters the Assumptions in the following manner: The current Assumption maps a primitive Structure onto an HTML element. All Assumptions that have not been considered and that have the same binding are deleted from the Assumption list (see image 5.2).

Evaluation Using the assertions made in the previous phase (direct tests) the occurrence of correctly filled out primitive and combined Structures. **Structure Assertions** are made to keep track of which Structures are thought to be mapped correctly. In this phase also tests for relative location are done. The thought behind this is that combined Structures have a location dependency - they can be found next to each other or above each other. After all assumptions are tested, the Configurations list is reordered in the next phase (pushUp Configs, see figure 5.3).

PushUp Configs The Structure assertions are used to reorder the Configurations in the Configuration list. This is done by means of the `configPushUp` method offered by the HTML Framework. It works as described below:
facts:

1. The current Configuration is a rewritten combined Structure.

2. Each Structure assertion indicates that a certain structure. was found.

All untested Configurations are being checked for combined Structures that were rewritten in the same way as the asserted combined Structures. For each such found Structure, its containing Configuration receives one point. The unchecked Configurations are then reordered based on the number of points. This way the Configuration list will have those Configurations bubbled to the top that have the most matching Structures.

After this phase the next Configuration is selected. In the next section, a run of this agent is described.

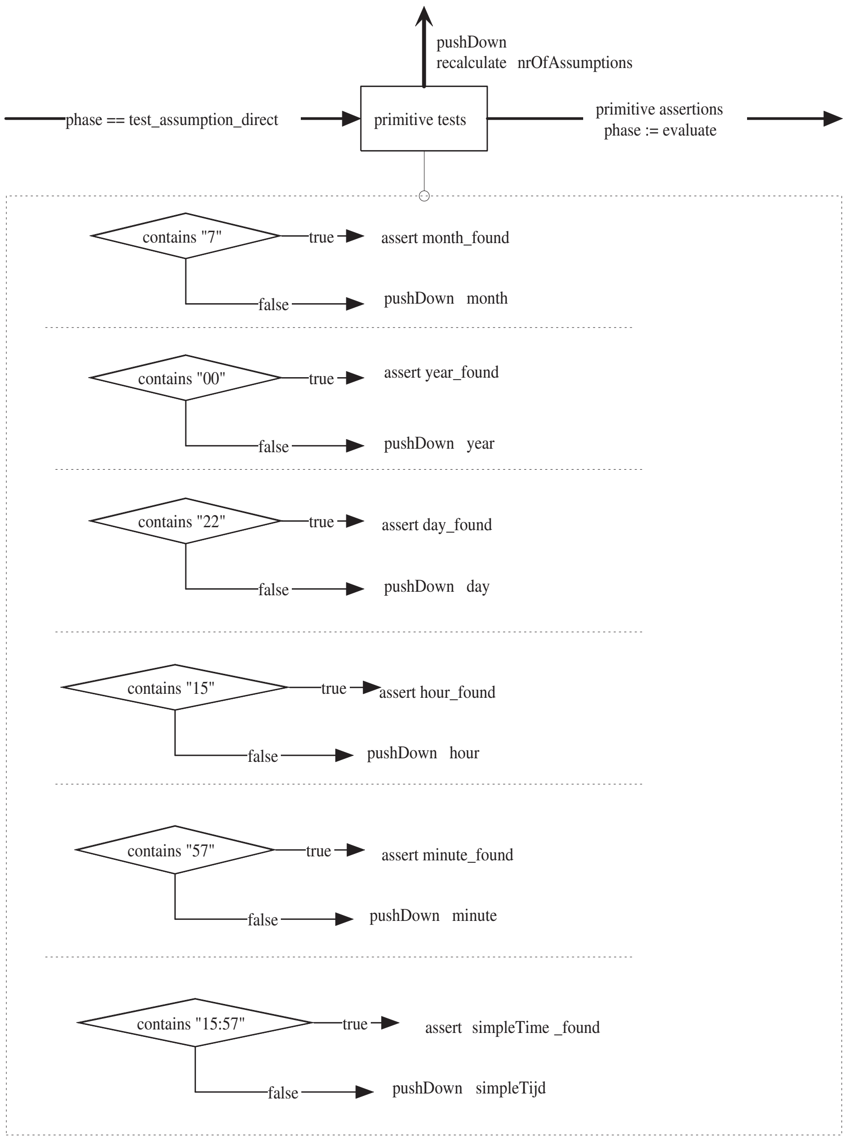


Figure 5.2: Flow diagram of direct tests of the push agent

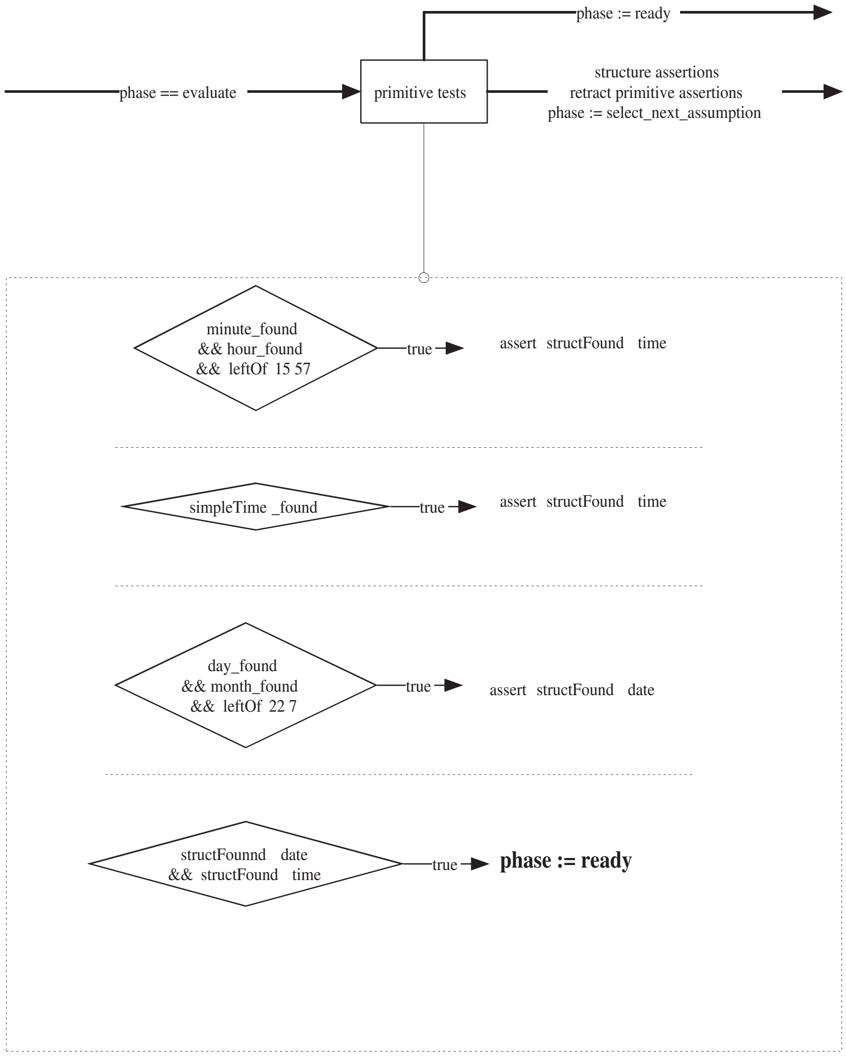


Figure 5.3: Flow Diagram of Indirect Tests of the Selective Agent

5.6 Testing the push agent

5.6.1 Testing environment

A testing application has been defined that requests a date and a time. If date or time are provided to the application correctly, it reproduces them. If not, it gives an error message. The application is programmed as a Java Server Page (see http://java.sun.com/products/jsp/). A JSP is a combination of HTML code and a Java program. With JSPs, HTML interfaces can be programmed fairly easily.

The date is requested as two numbers, the time is also requested in two separate fields. In figure 5.4 the request page can be seen within the HTML Framework. Furthermore, two dummy fields are added: one for year and one for name. Values filled in in one of these fields are ignored by the program. The error message that the testing application provides is the number "-1" filled out in an input field where in case a field was filled out correctly, the correct value would appear.

An error value is produced if

- The value for month is not an integer between 0 and 13.
- The value for day is not an integer between 0 and 32.
- The value for hour is not an integer between -1 and 24.
- The value for minute is not an integer between -1 and 60.

We load our HTML Framework with the following Structures:

```
Structure(date_time) {
    type = options;
    options = {(date,time)};
}

Structure(tijd) {
    type = options;
    options = {simpleTime, combinedTime};
}

Structure(combinedTime) {
    type = options;
    options = {(hour,minute)};
}

Structure("simpleTime") {
```

```
    type = input;
    value = "12:57";
}

Structure("hour") {
    type = input;
    value = "15";
}

Structure("minute") {
    type = input;
    value = "57";
}

Structure("date") {
    type = options;
    options = {simpleDate, (month,day)};
}

Structure("simpleDate") {
    type = input;
    value = "28.06";
}

Structure("month") {
    type = input;
    value = "07";
}

Structure("day") {
    type = input;
    value = "22";
}
```

In “natural language” these Structures would define that a date is either filled out in one (28.06) or or in two fields (22 for day and 07 for month). The same holds for time (12:57 or 15 for hour and 57 for minute respectively).
Our agent is as described in the previous section. The following subsection will walk through the steps the agent takes.

5.6.2 Test run

In this section the output of the agent is analyzed in order to show which steps it has taken to find which input fields on the date and time interface are which. After several startup

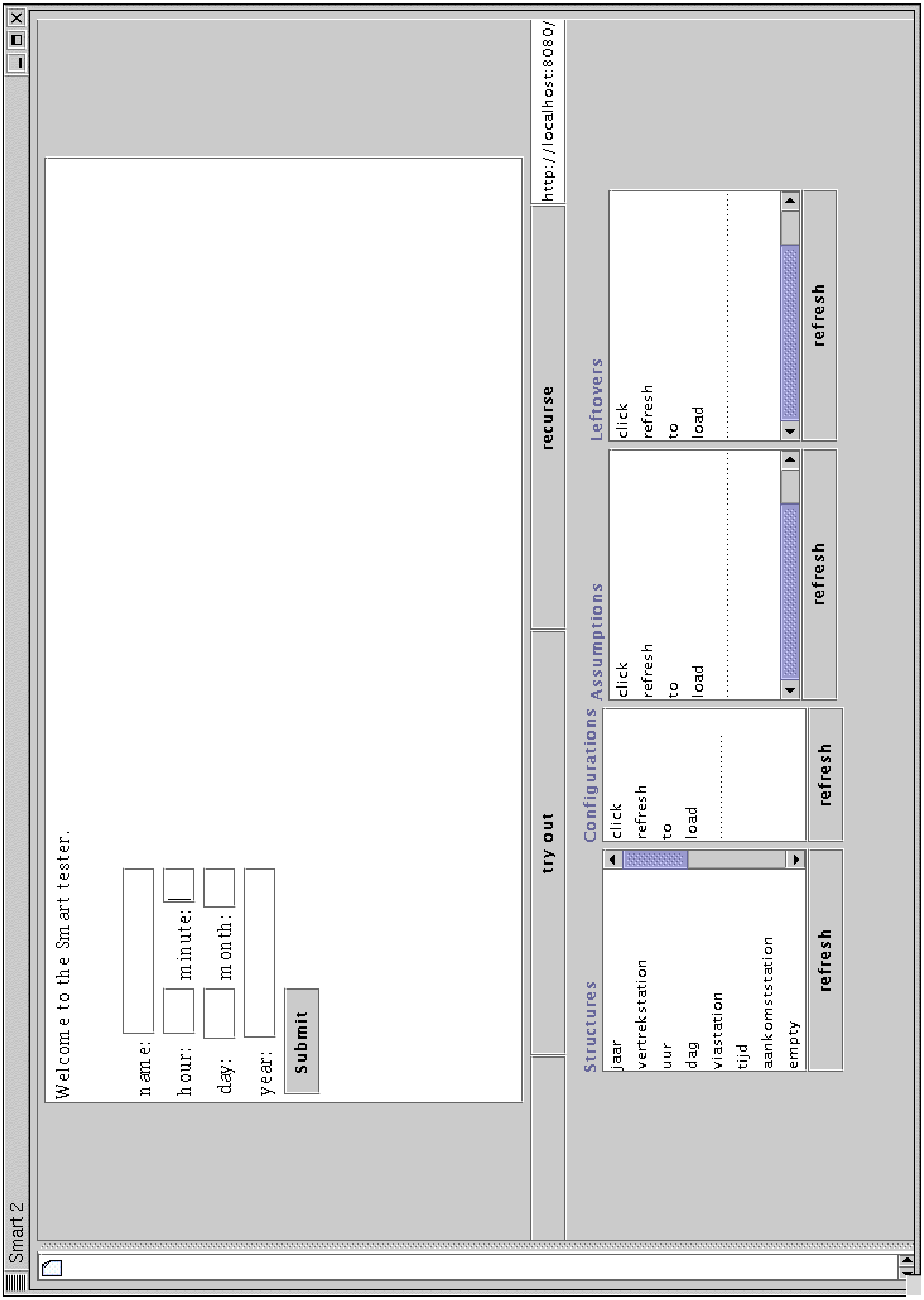


Figure 5.4: Starting page of the Time of Birth testing application

messages, the agent indicates it is searching for date_time (line 3). Afterwards it chooses the first Configuration that the HTML Framework offers for this Structure ([day, month, simpleTime]). It then calculates all possible Assumptions for the starting page. Only 4 of the 120 possibilities are shown here. Each Assumption consists of field-value pairs. In the first Assumption the hour input field is related to the day Structure, the input field for day is related to the month Structure and the input field for year is related to the simpleTime Structure.

```
 1 Jess, the Java Expert System Shell
 2 Copyright (C) 1998 E.J. Friedman Hill and the Sandia Corporation
 3 Jess> date_time
 4 cfg done
 5 [day, month, simpleTime]
 6 [300{hour=day, day=month, childname=simpleTime},
 7 300{hour=day, day=month, year=simpleTime},
 8 ...<116 intermediate combinations are left out>
 9 300{childname=simpleTime, month=month, minute=day},
10 300{month=month, year=simpleTime, minute=day}]
```

Then the Leftovers are constructed by the Framework. One Leftover is found, consisting of the fields year, month and minute. They are filled in with an empty value.

```
11 year=&
12 year=&month=&
13 leftover: year=&month=&minute=&
```

On line 14 the parameters with which our testing application is called are shown. The lines immediately after (line 15 and 16) shows that our agent was not satisfied with the result. It didn't find minute and takes pushdown action, so it calls the interface's pushDown procedure telling it to remove all Assumptions that have the same binding for minute. Since minute is unbound here, the action will not sort any effect. the same holds for pushing down the Assumptions with hour bound.

```
14 params: childname=12:57&day=07&hour=22&year=&month=&minute=&
15 pushdownUsingcurrent minute null 300{hour=day, day=month,
    childname=simpleTime}
16 minute not found 1
17 pushdownUsingcurrent uur null 300{hour=day, day=month,
    childname=simpleTime}
18 uur not found1
```

The pushDown mechanism works for the Assumptions binding simpleTime to childname. Line 20 indicates that all Assumptions from 2 to 120 (the last one) are checked for the same binding, and removed in case they have this binding.

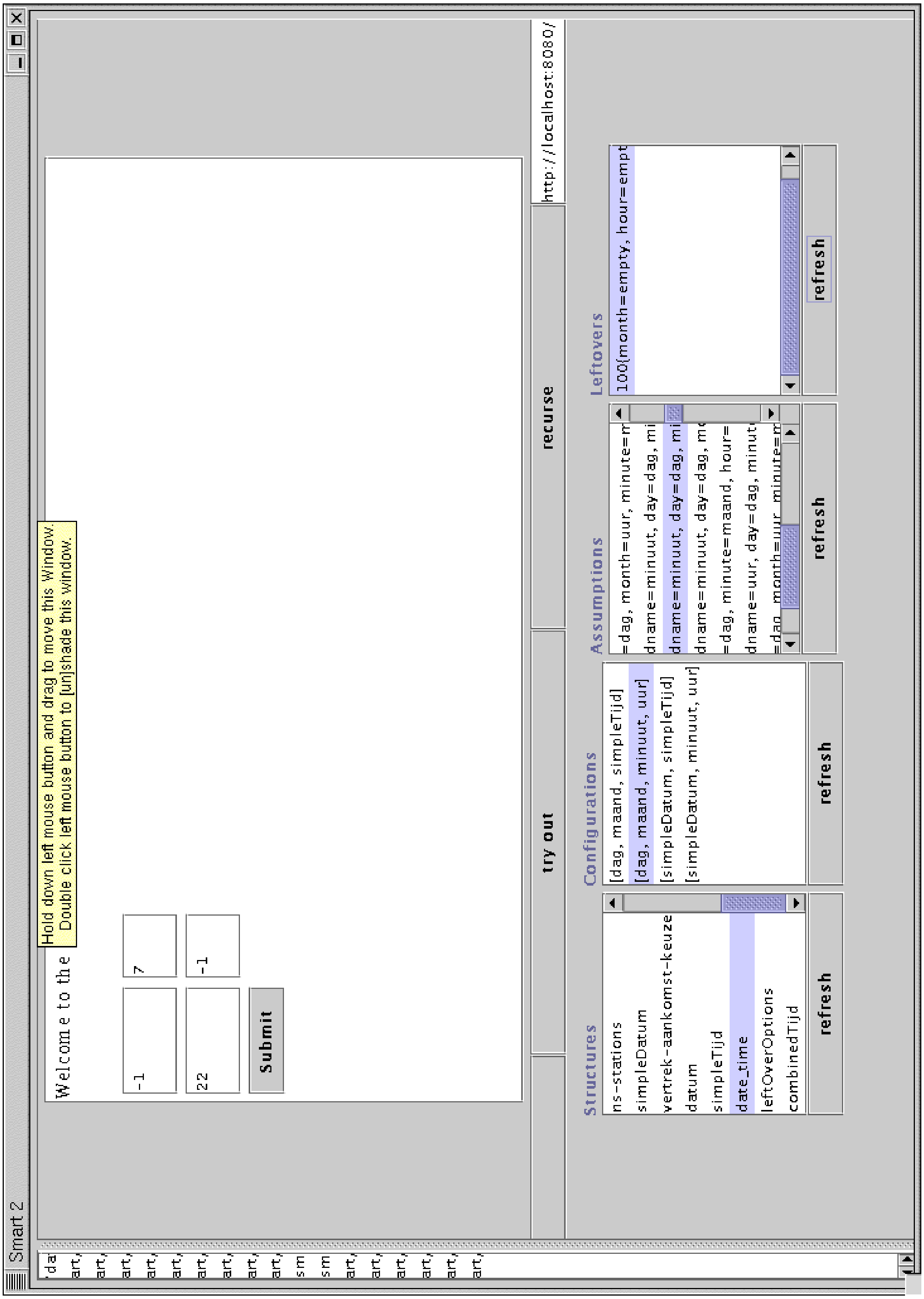


Figure 5.5: An intermediate result.

```
19 pushdownUsingcurrent simpleTime childname 300{hour=day, day=month,
    childname=simpleTime}
20 pushdown from 1 to 120
21 simpleTime not found1
```

The next 2 lines (22 and 23) show that values for day and month were found (i.e. occur on the result page). It is then checked if day occurs left of month (line 24).

```
22 day found
23 month found
24 resultevaluator: leftof 22 7
```

The evaluation phase starts here, but nothing of interesting is found. The next Assumption is selected and another checking round is started.

```
25 switching to evaluate
26 retracted month_found
27 retracted day_found
28 switching to check_config
```

Some steps were left out here, the first interesting step is where another set of Assumptions is removed. From the previous pushDown only 101 Assumptions were left (line 38). All Assumptions from 2 to 101 are checked.

```
37 pushdownUsingcurrent simpleTime year 300{hour=day, day=month,
    year=simpleTime}
38 pushdown from 2 to 101
39 simpleTime not found2
```

This testing lasts five times, after that no Assumptions are left for testing, the next Configuration is selected (lines 119 to 121). Again all Assumptions are calculated (lines 121 to 127)

```
118 switching to check_config
119 switching to select_next_config
120 [day, month, minute, hour]
121 [400{hour=day, day=month, childname=minute, year=hour},
122 400{hour=day, day=month, childname=minute, month=hour},
123 400{hour=day, day=month, childname=minute, minute=hour},
124 ...<354 intermediate combinations are left out>
125 400{hour=hour, month=month, year=minute, minute=day},
126 400{day=hour, month=month, year=minute, minute=day},
127 400{childname=hour, month=month, year=minute, minute=day}]
```

Then, after nine rounds of testing Assumptions and pushing down unlikely candidates, the fifteenth tested Assumption meets all the tests: day, month hour an month occur on the page (lines 280 to 283), 15 appears left of 57 and 22 appears left of 7 (lines 284 to 289). The agent concludes that time and date were found and the stop condition is reached.

```
switching to check_config
childname=&
leftover: childname=&year=&
params: hour=15&minute=57&month=07&day=22&childname=&year=&
pushdownUsingcurrent simpleTime null 400{day=day, hour=hour,
  month=month, minute=minute}
simpleTime not found9
day found
month found
minute found
hour found
resultevaluator: leftof 15 57
matched left: 15
matched right: 57
resultevaluator: leftof 22 7
matched left: 22
matched right: 7
switching to evaluate
time found
date found
ready
```

The situation in which the Framework is left after the agent has stopped is depicted in figure 5.6. The complete transcript can be found in appendix B.

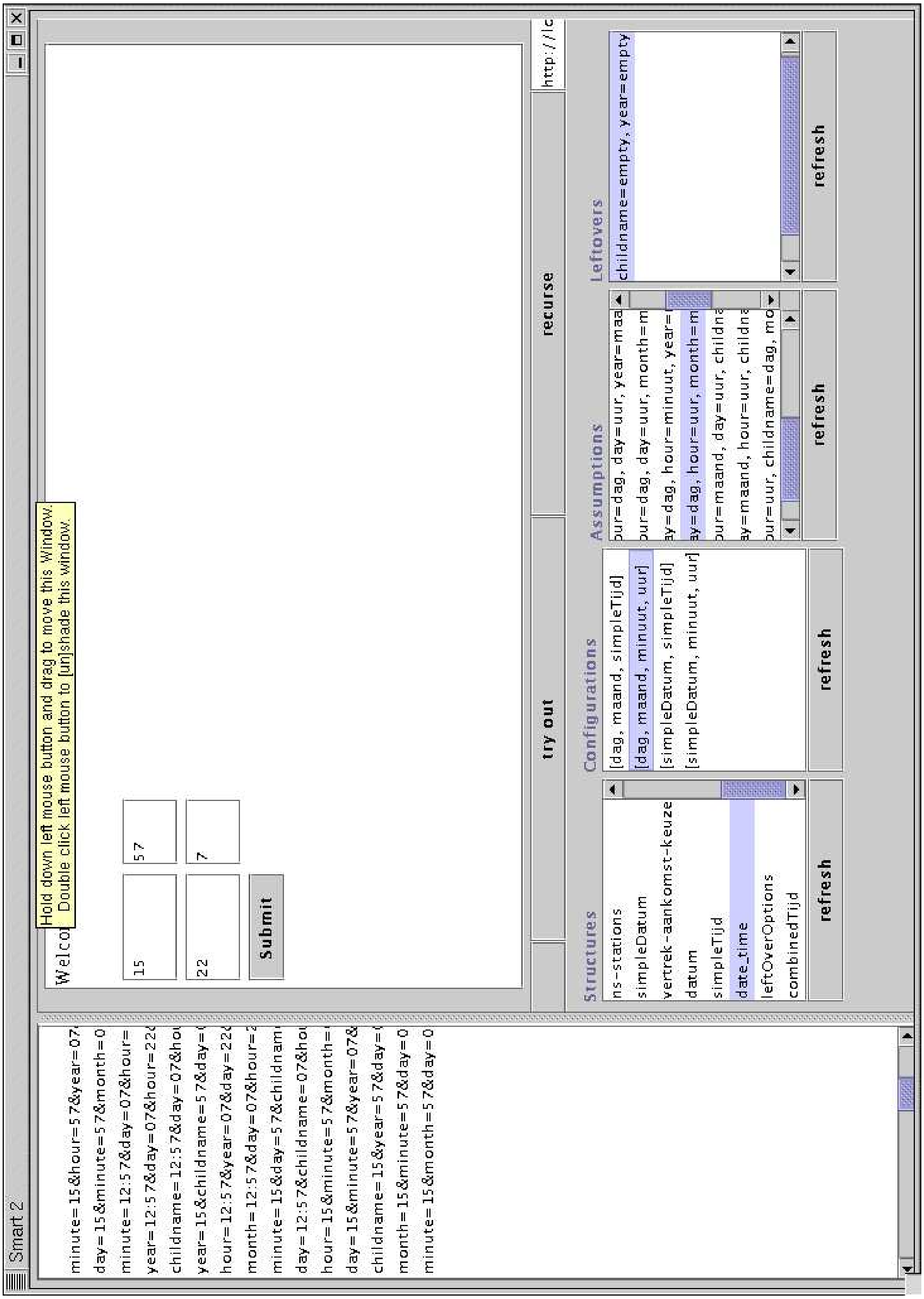


Figure 5.6: Final page of the Time of Birth testing application. A labeling was found.

Chapter 6

Conclusions and recommendations

Based on the idea of the KISS method that an organizational model can be derived by interacting with the organization, we have have worked on automatially obtaining a user interface model from interacting with the user interface. In this thesis a system was described that does this by extracting interface semantics through determining the meaning of input fields of interfaces. Agent technology was used to approach the user interfaces.

In order to make the extraction of semantics feasible, we have restricted our research to Internet based rail travel planners. Such planners have a start page and a result page. The start page contains an HTML form with which an itinerary can be specified. The result page consists of a travel schedule. If a start page is not filled out correctly, a rail planner offers a new form with which a user can retry or supplement his itinerary. These pages are called “intermediate pages”.

The work described in this thesis has led to the following results:

- A corpus of seven rail planners was created.
- We modeled human computer interaction based on the analysis of these planners.
- A notation that can be used to describe start pages based on their functionality was defined.
- A framework to use the start page description language to fill out a start page form and retrieve the results was created.
- A testing module for testing result pages was implemented.
- Two example agents for extracting user interface semantics were developed. These agents are programmed as expert systems that have rules to take action based on intermediate and result pages.
- A testing environment was created.

From the results the following conclusions can be drawn:

Conclusion 1: It has been shown that in principle it is possible to extract semantics from user interfaces by using an agent.

Conclusion 2: Automating the evaluation of the interaction based on the specification is still an obstacle when extracting user interface semantics. We have found that if itineraries were incompletely or erroneously specified, the response from the information system is in many cases unrelated to the start page specification.

Conclusion 3: Because of the large variety in possible HTML interfaces, it is difficult to build an agent that is effective in extracting interface semantics, even with the limited case of rail planner interfaces.

Recommendations There are several ways to proceed with this research. The work described in this thesis was done as part of the idea that offering a unified interface to information systems with common functionality is useful. This idea was presented in the introduction. Only a small part of the research and implementation necessary for a successful agent based approach has been done.

Useful directions to take seem to be:

- Define a notation with which the interaction with information systems can be described. A specification of an information system with such a notation could ultimately be used to generate expert rules for the interaction with the system.
- Instead of an expert system based agent, other AI techniques such as case based reasoning could be applied for the extraction of semantics.
- Information systems other than rail travel planners, such as rental car reservation systems could be tried.
- The bridge to user interface description languages could be made in order to express the semantics found during the interaction.
- The results of automatic semantics extraction can be used as metrics for e.g. ease of use and functionality.

Appendix A

Rail planner data

This chapter shows the screendumps of seven rail planners. Each of the planners was used to retrieve a time table of a trip from a local city to another local city. For all planners the initial screen is shown, then one or more intermediate screens, a timetable and if possible, an error page.

A.1 Netherlands

`http://www.ns.nl/cgi-bin/nsi.exe/vraag`

A.1, A.2, A.3.

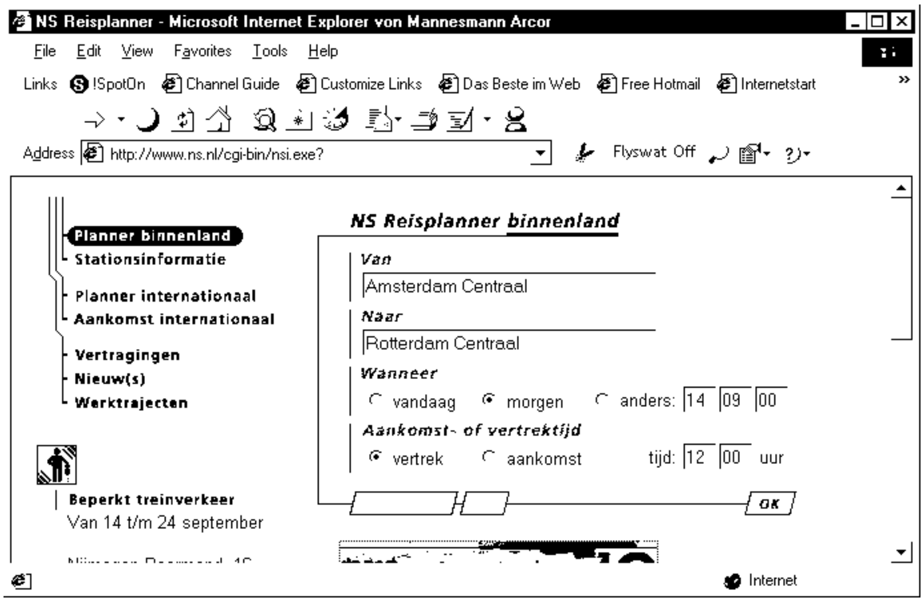


Figure A.1: Start page

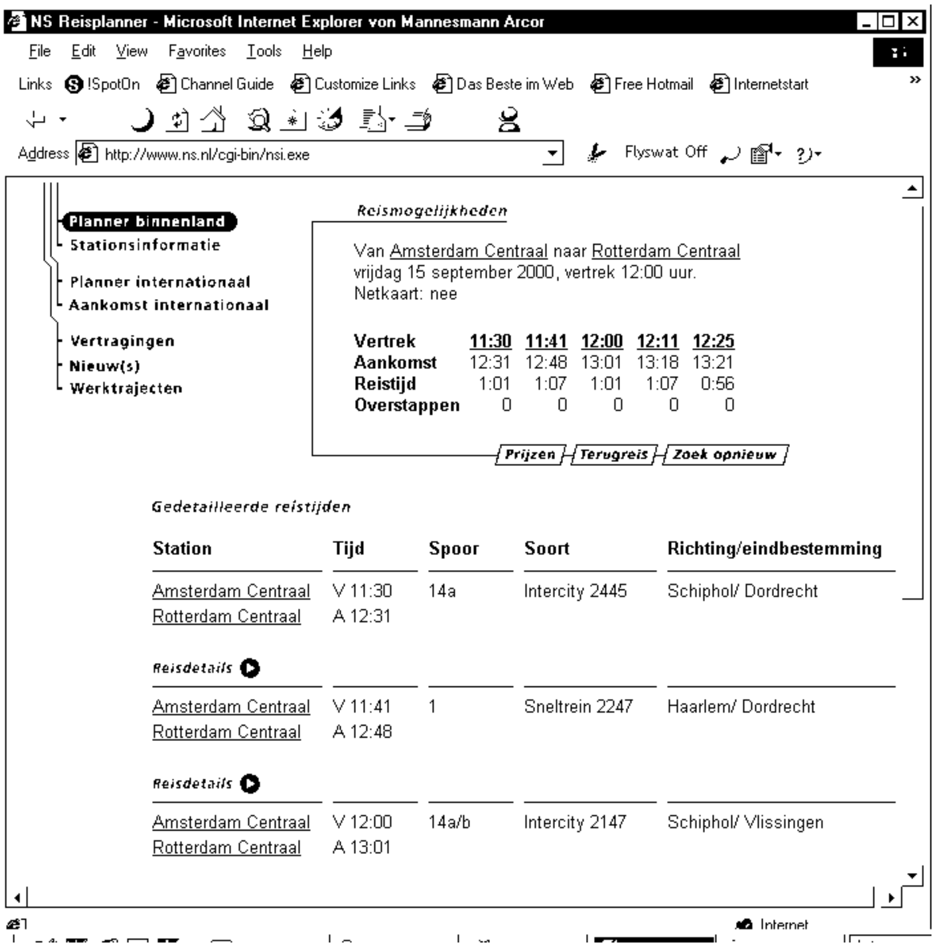


Figure A.2: Result of query: from Amsterdam Centraal to Rotterdam Centraal, departing tomorrow at 12:00

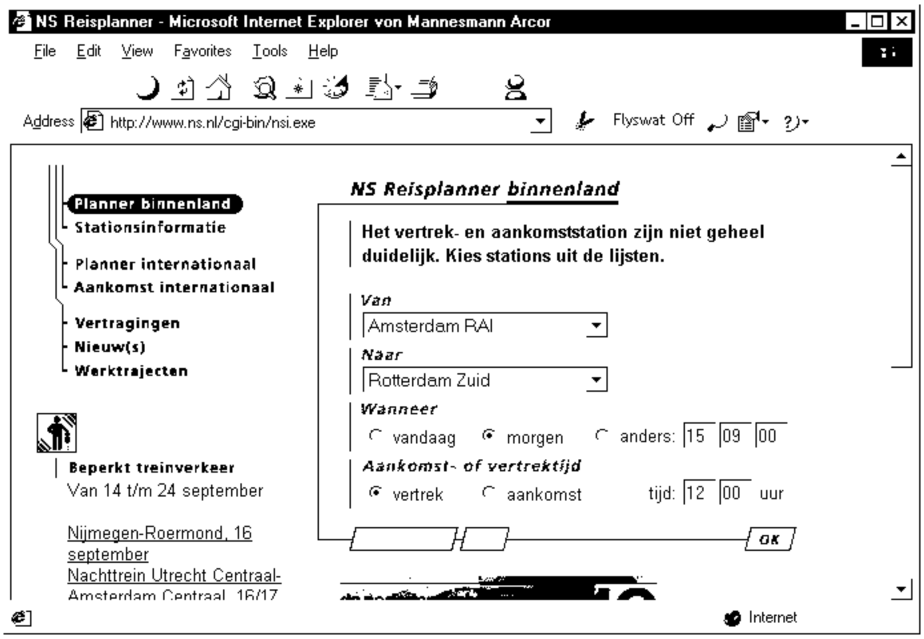


Figure A.3: Incomplete page, result of query: from Amsterdam to Rotterdam, departing tomorrow at 12:00

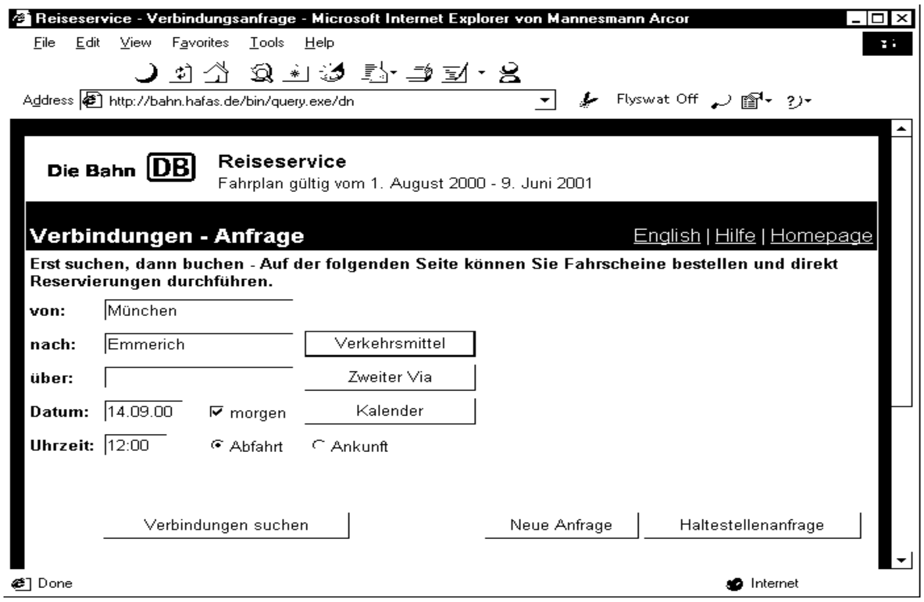


Figure A.4: Start page

A.2 Germany

`http://bahn.hafas.de/bin/query.exe/dn`

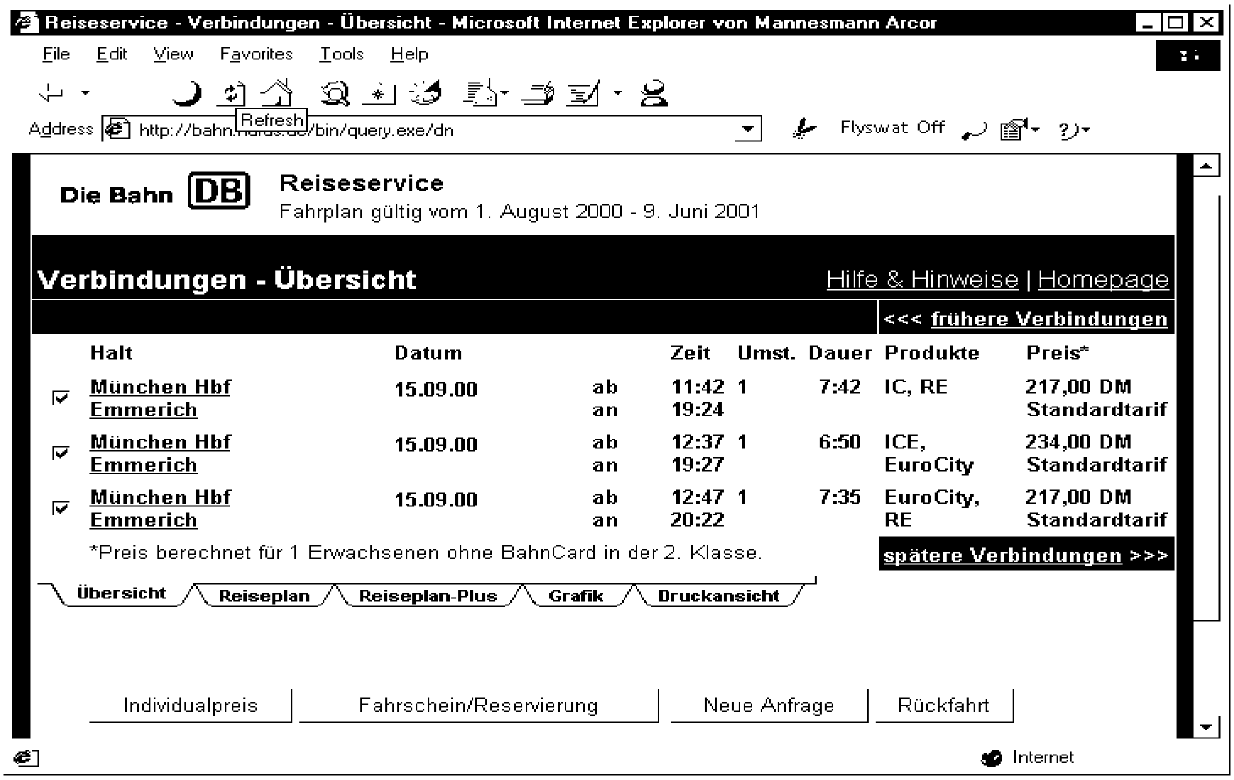


Figure A.5: Result of query: from Munich to Emmerich, departing at 12:00

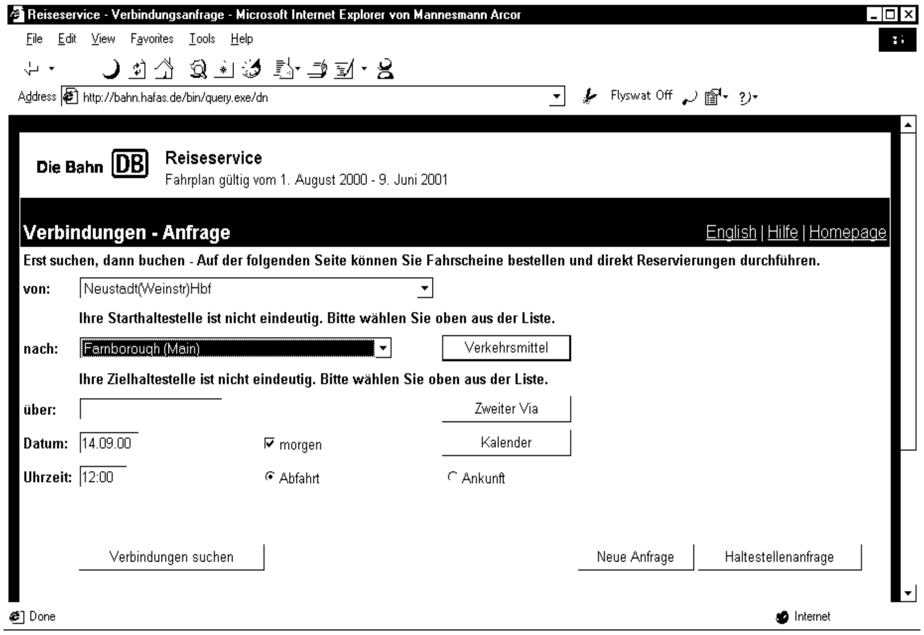


Figure A.6: Incomplete page, result of query: from Farnkfrut to Keinstadt, departing tomorrow at 12:00

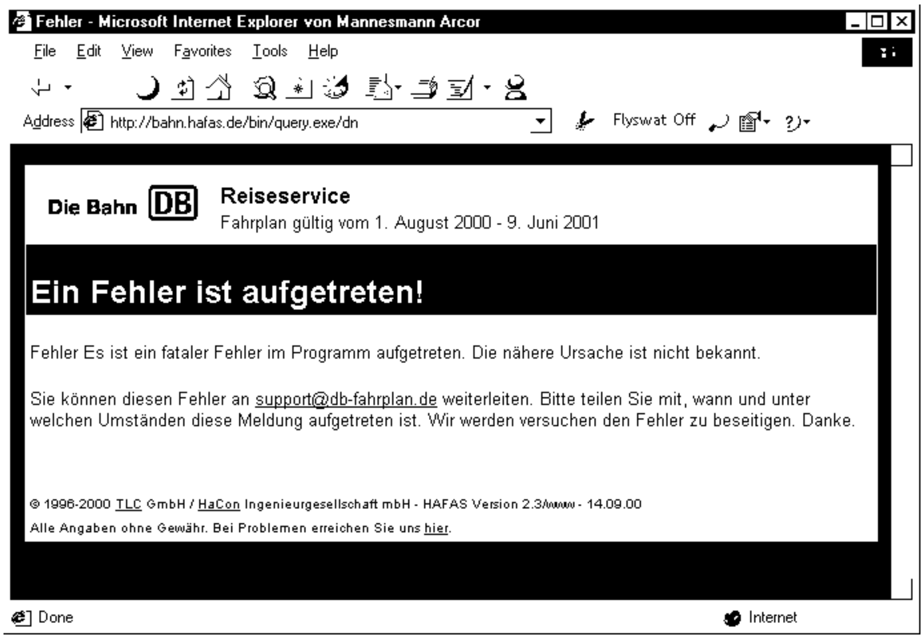


Figure A.7: Error page

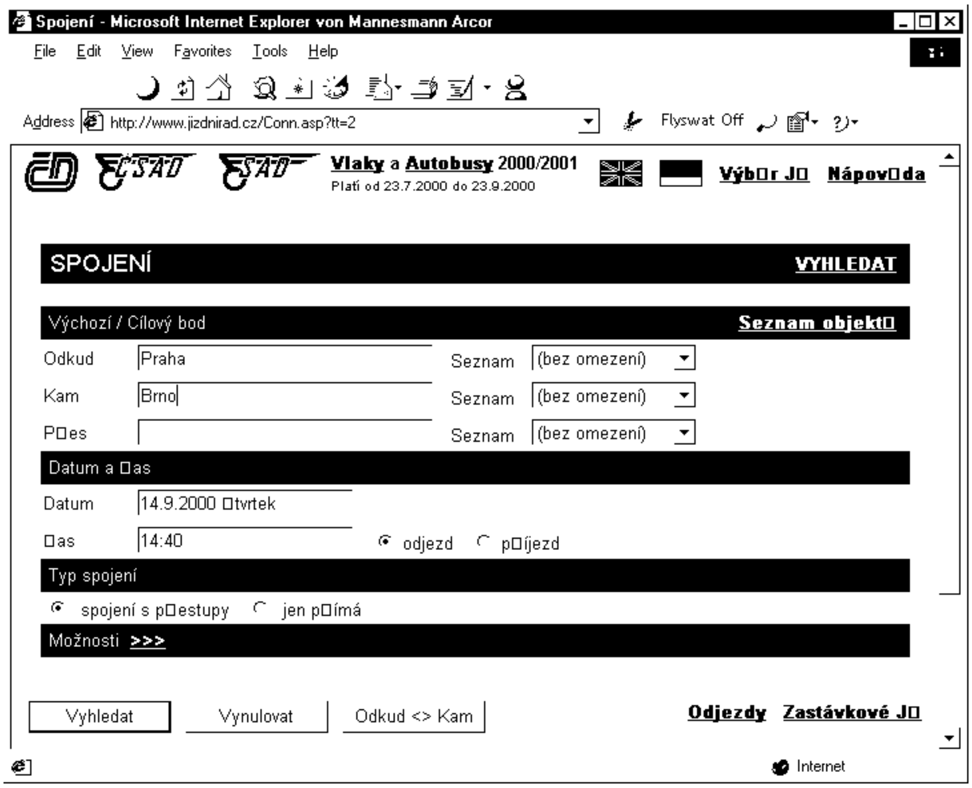


Figure A.8: Start page

A.3 Czech Republic

http://www.jizdnirad.cz/Conn.asp?tt=2

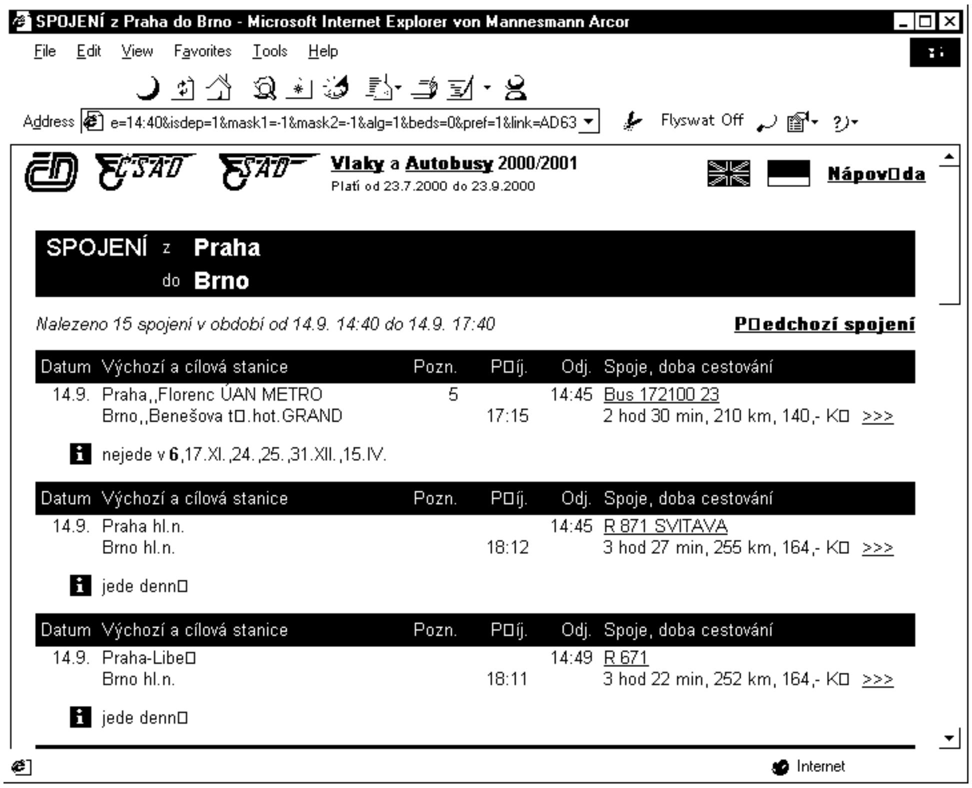


Figure A.9: Result of query: from Praha to Brno, departing at 12:00

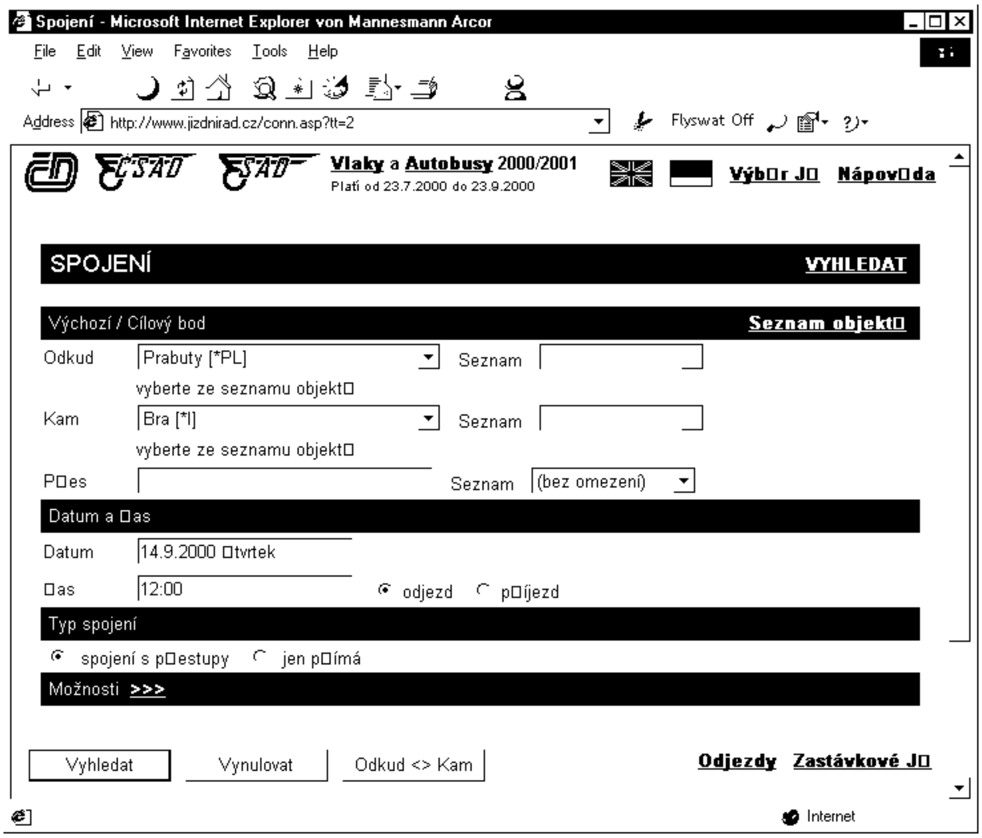


Figure A.10: Incomplete page, result of query: from Pr to Br, departing at 12:00

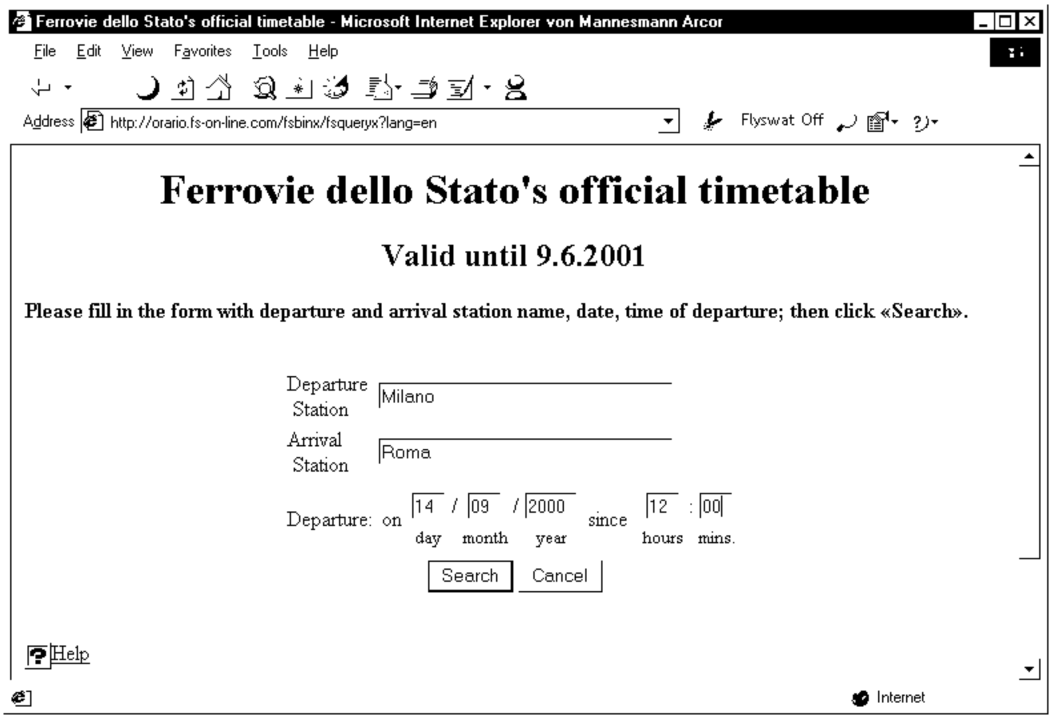


Figure A.11: Start page

A.4 Italy

`http://orario.fs-on-line.com/fsbinx/fsqueryx?lang=en`

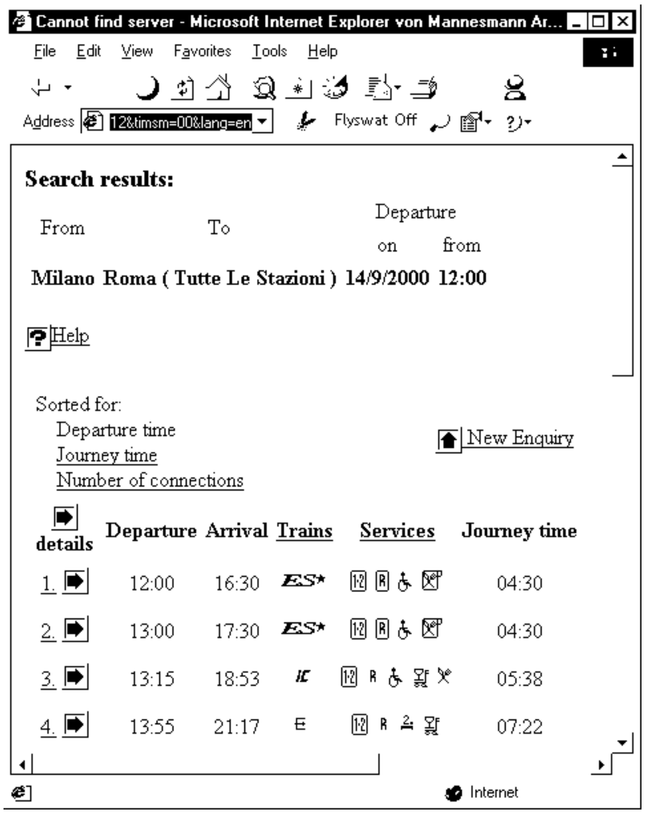


Figure A.12: Result of query: from Milano to Roma, departing at 12:00

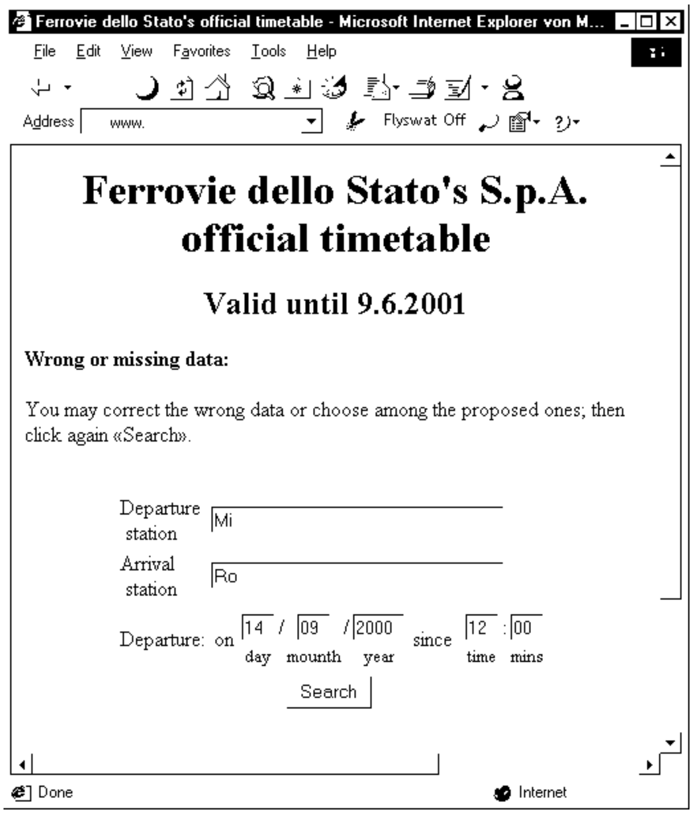


Figure A.13: Incomplete page, result of query: from Mi to Ro, departing at 12:00

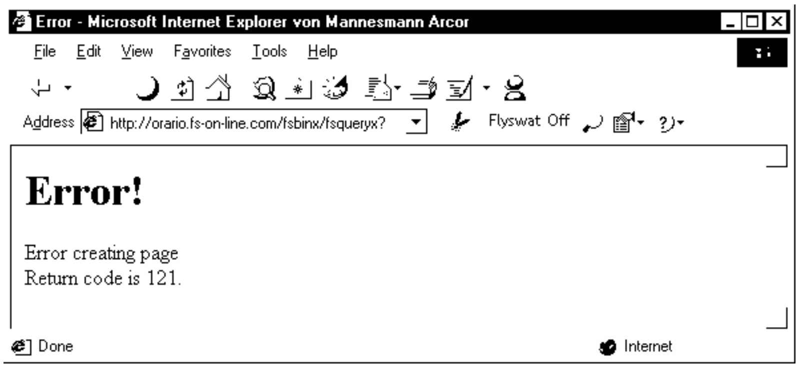


Figure A.14: Error page

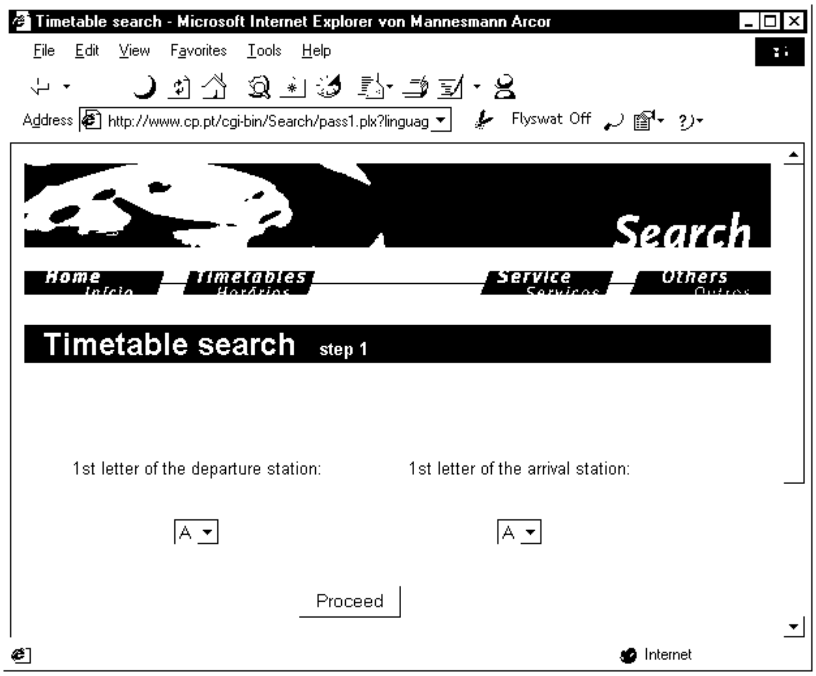


Figure A.15: Start page

A.5 Portugal

`http://www.cp.pt/cgi-bin/Search/pass1.plx?linguagem=uk`

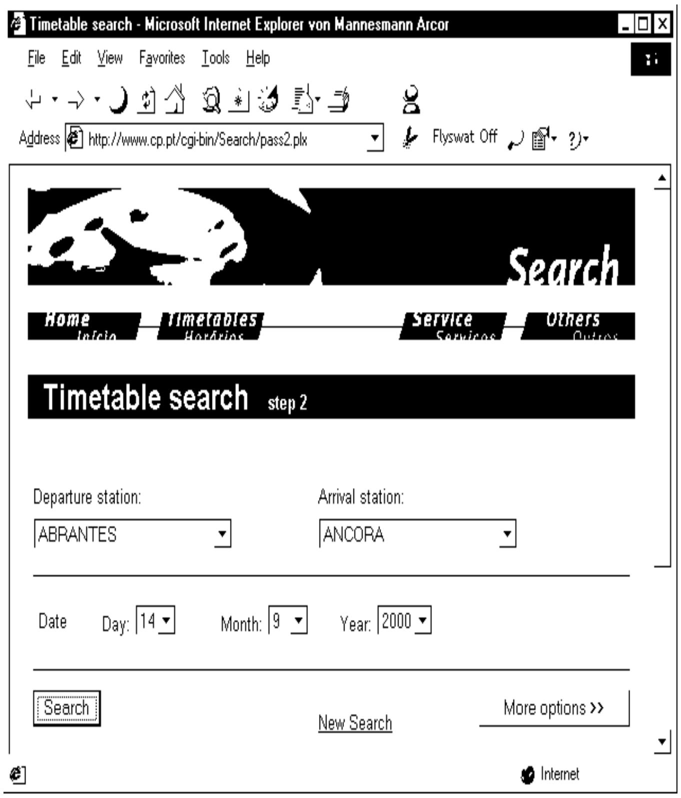


Figure A.16: Intermediate page after selecting "A" for the first letter of both departure station and arrival station

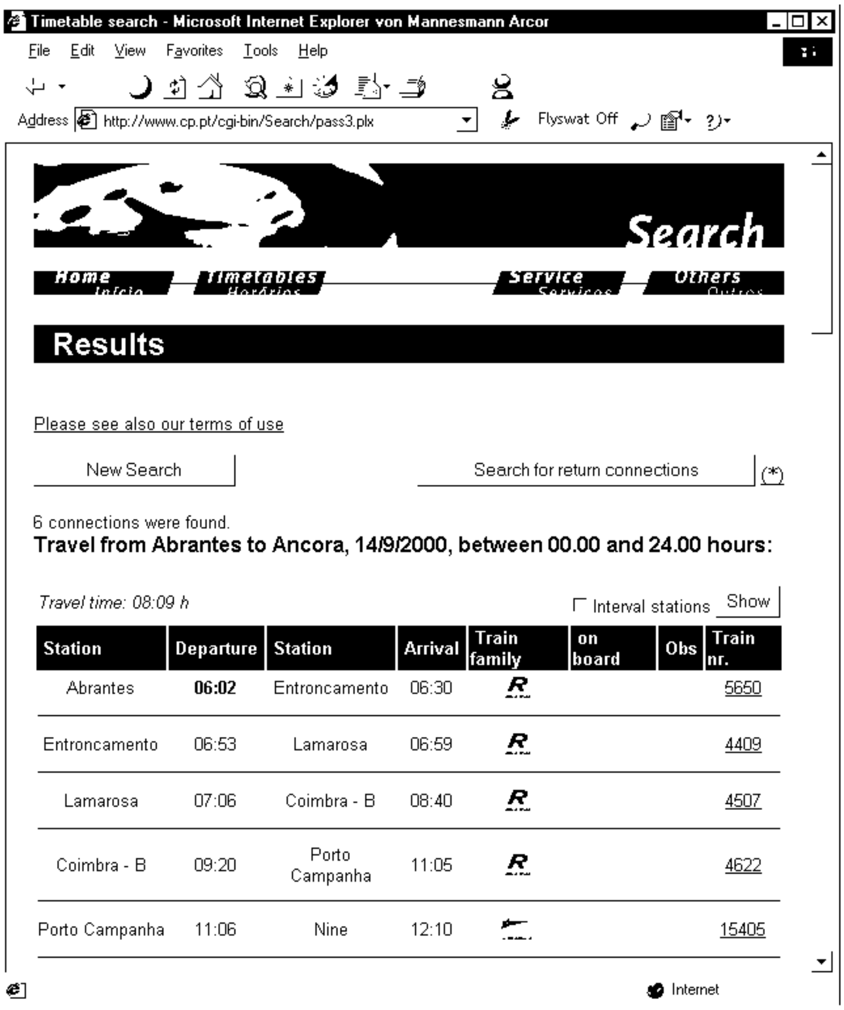


Figure A.17: Result of selecting Abrantes and Ancora. The result shows all connections of the default day (today)

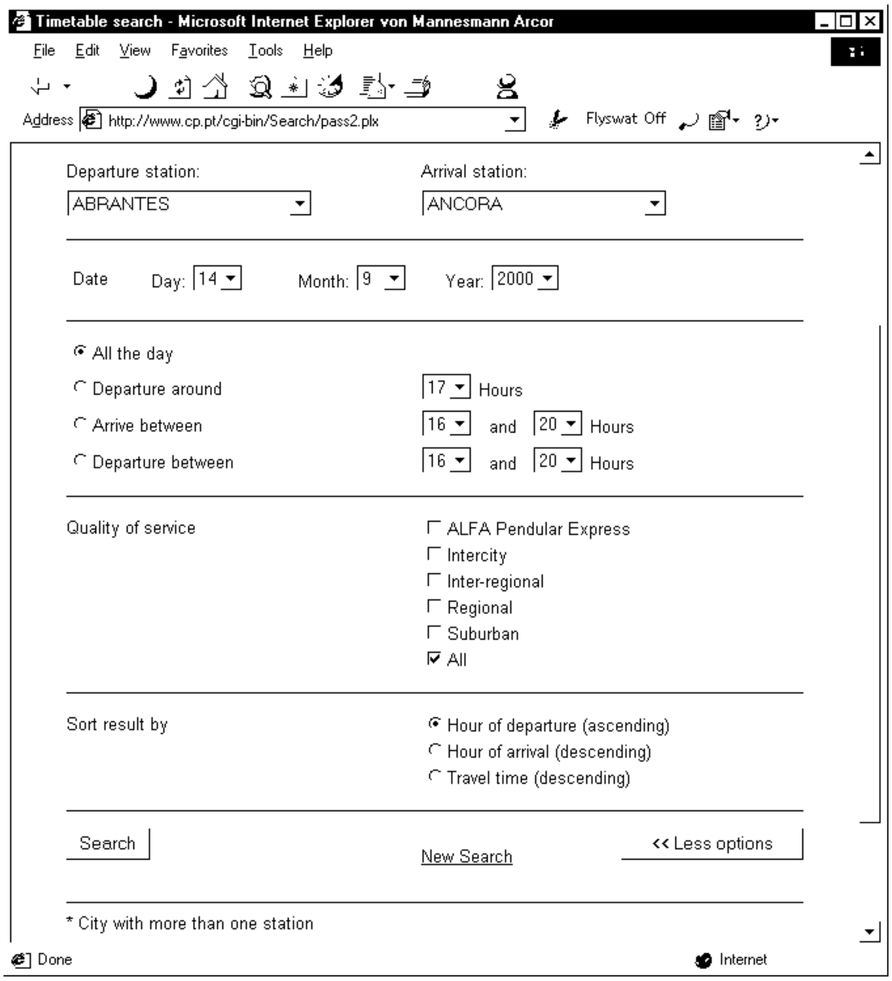
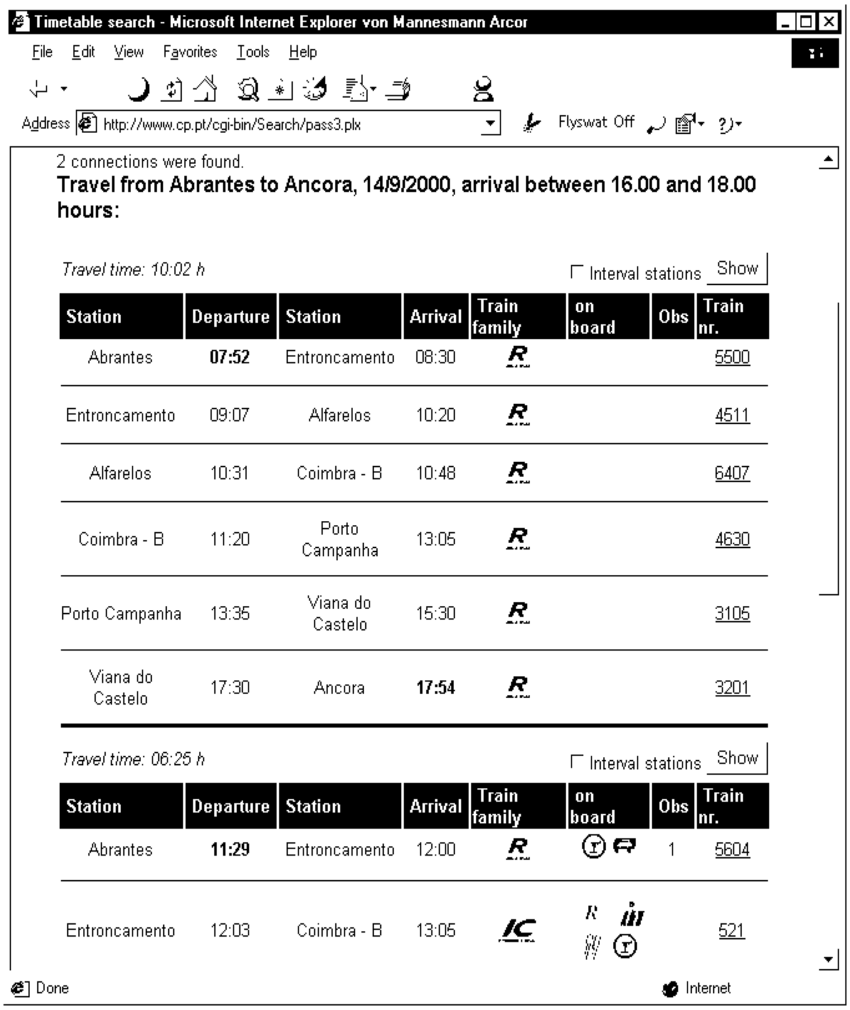


Figure A.18: Intermediate page after selecting more options in Figure A.16


Timetable search - Microsoft Internet Explorer von Mannesmann Arcor

File Edit View Favorites Tools Help

Address http://www.cp.pt/cgi-bin/Search/pass3.plx

Flyswat Off

2 connections were found.

Travel from Abrantes to Ancora, 14/9/2000, arrival between 16.00 and 18.00 hours:

Travel time: 10:02 h Interval stations Show

Station	Departure	Station	Arrival	Train family	on board	Obs	Train nr.
Abrantes	**07:52**	Entroncamento	08:30	R			5500
Entroncamento	09:07	Alfarelos	10:20	R			4511
Alfarelos	10:31	Coimbra - B	10:48	R			6407
Coimbra - B	11:20	Porto Campanha	13:05	R			4630
Porto Campanha	13:35	Viana do Castelo	15:30	R			3105
Viana do Castelo	17:30	Ancora	**17:54**	R			3201

Travel time: 06:25 h Interval stations Show

Station	Departure	Station	Arrival	Train family	on board	Obs	Train nr.
Abrantes	**11:29**	Entroncamento	12:00	R		1	5604
Entroncamento	12:03	Coimbra - B	13:05	IC	R		521

Done Internet

Figure A.19: Result of filling out "arrival between 16:00 and 18:00 as an additional option (see Figure A.18.

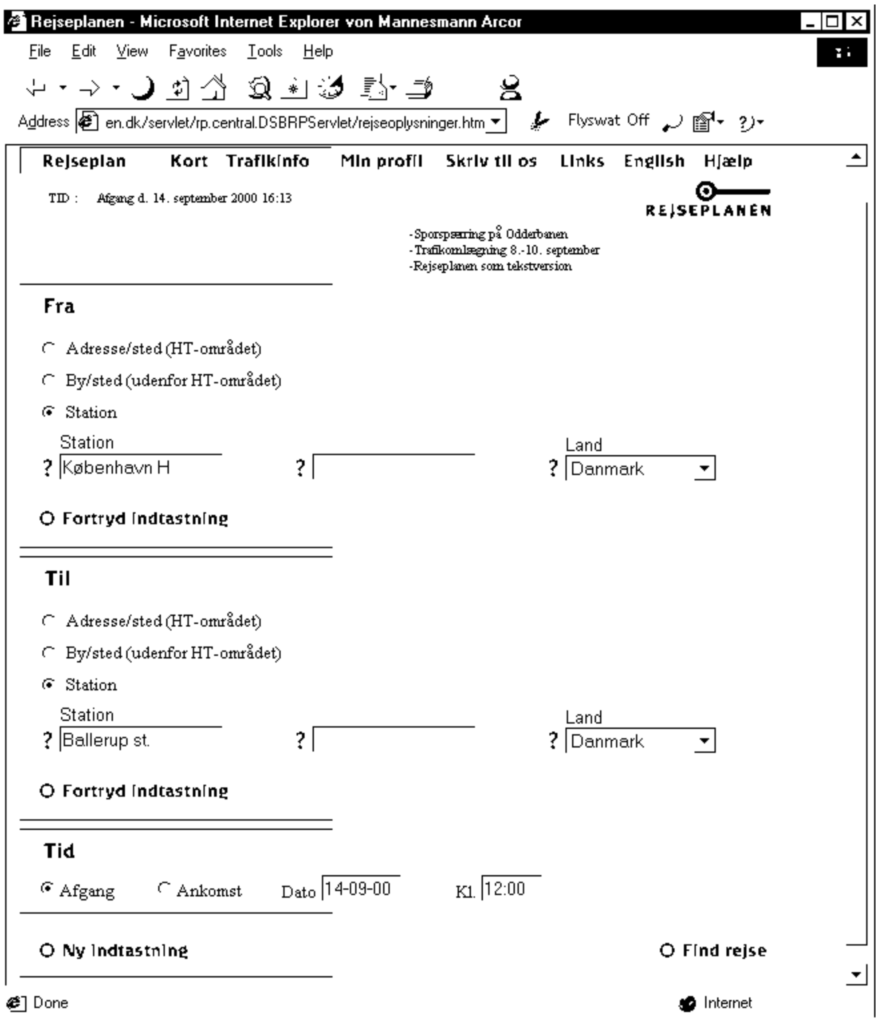


Figure A.20: Start page

A.6 Denmark

http://www1.rejseplanen.dk/servlet/rp.central.DSBRPServlet/
rejseoplysninger.htm

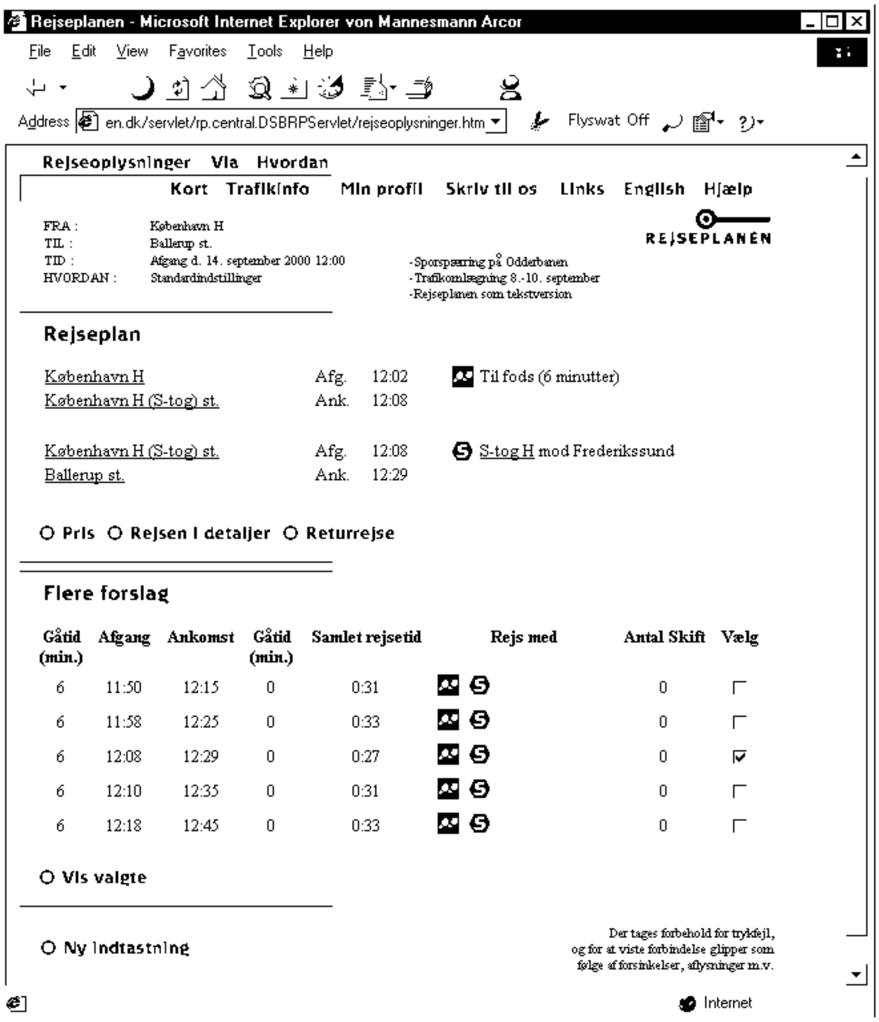


Figure A.21: Result of query: from Kobenhavn H to Ballerup st., departing at 12:00

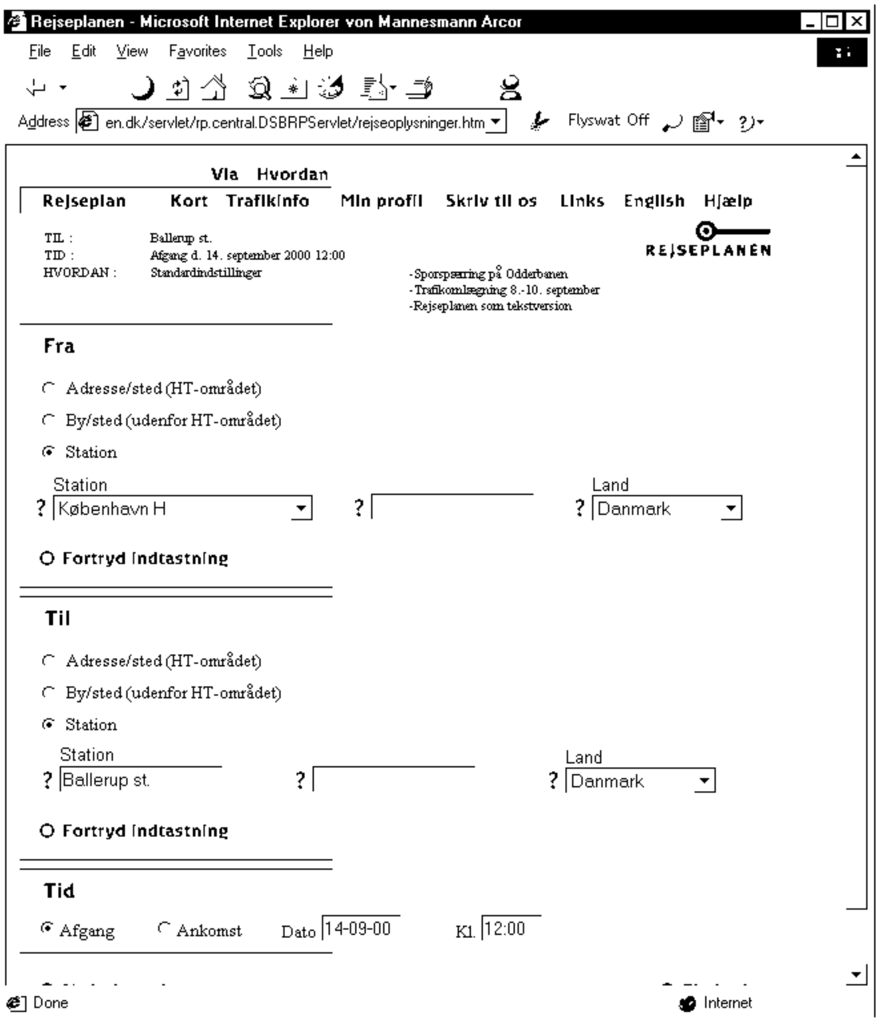


Figure A.22: Incomplete page, result of query: from Kopenhavn to Ballerup, departing at 12:00

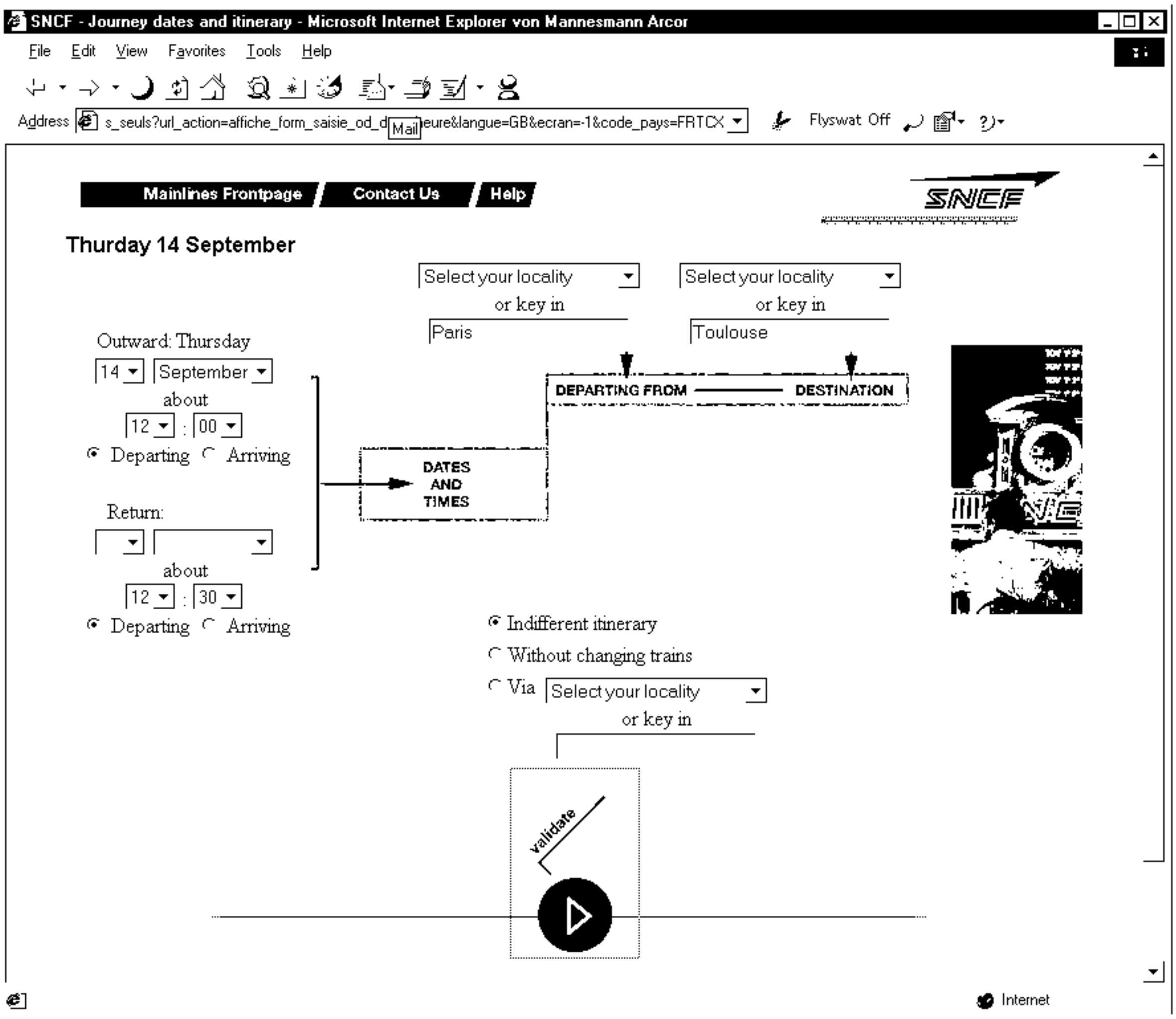


Figure A.23: Start page

A.7 France

`http://voyages.sncf.fr/reservation?ecran=-1&url_action=`

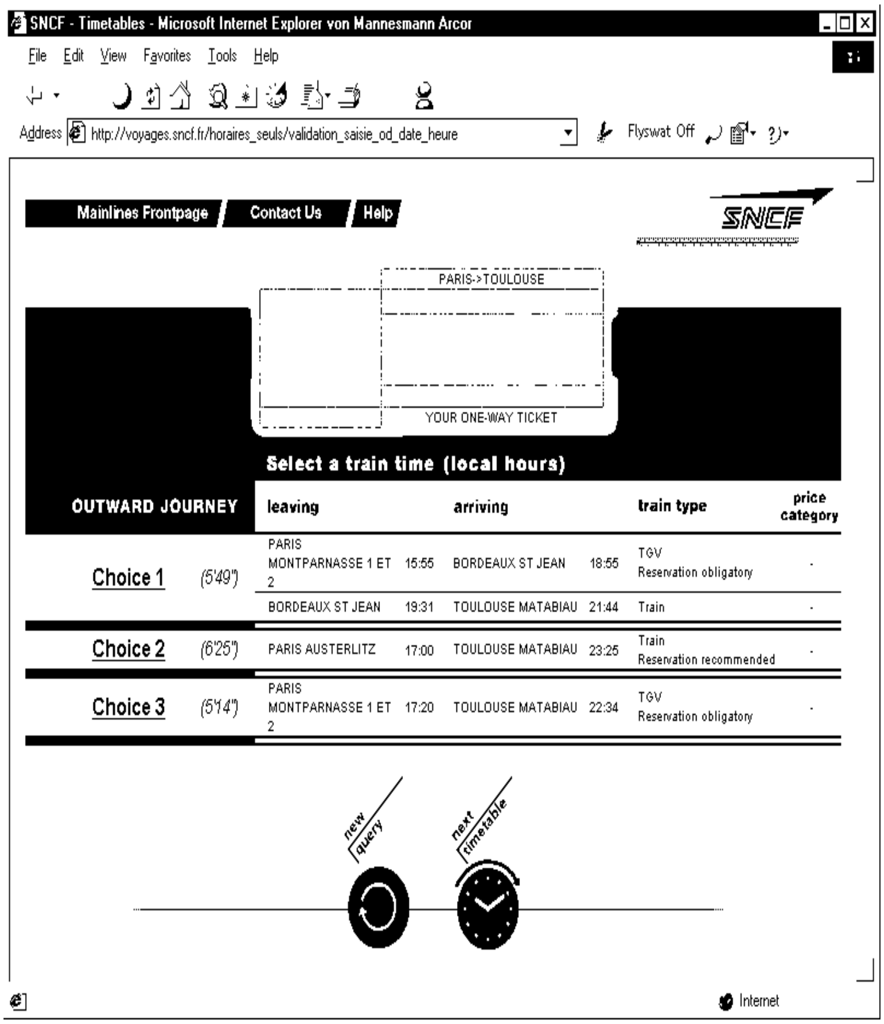


Figure A.24: Result of query: from Paris to Toulouse, departing at 12:00

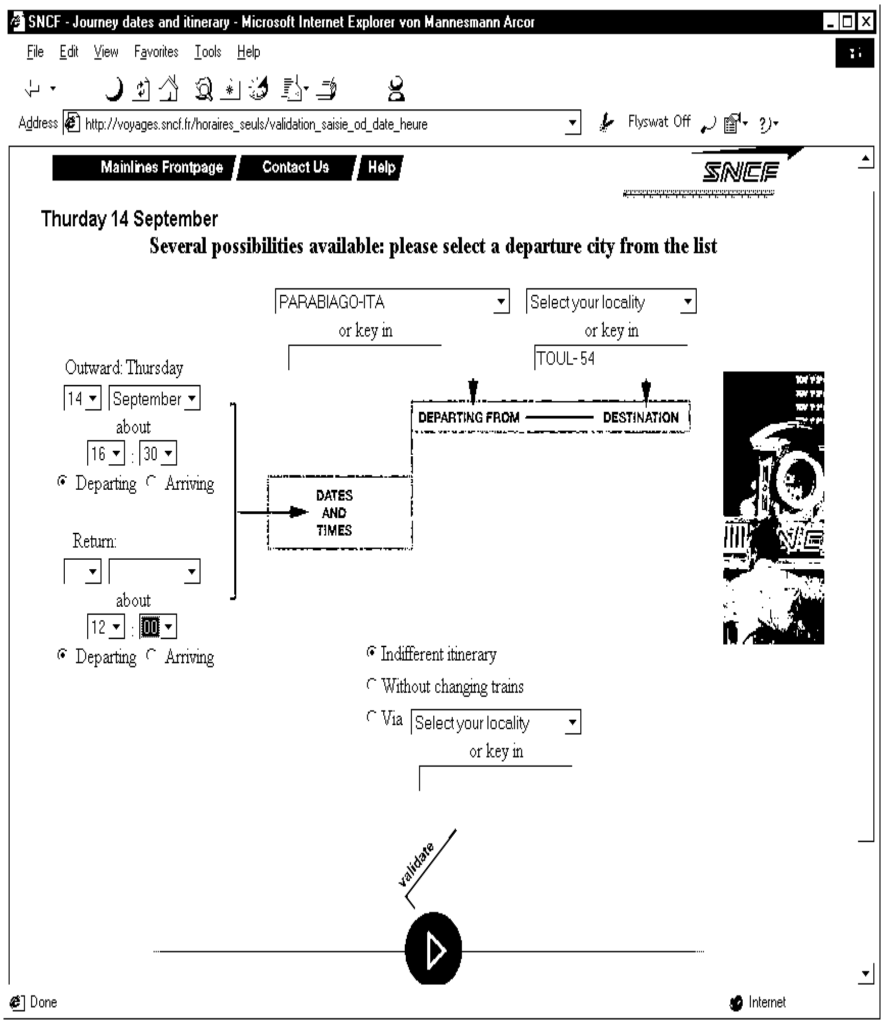


Figure A.25: Incomplete page, result of query: from "Par" to "Tou", departing at 12:00

Appendix B

Agent transcript

Transcript of the session of the pus agent with the test applications in chapter 5. The result of calculating all possible Assumptions has been abbreviated for space reasons.

```
Jess, the Java Expert System Shell
Copyright (C) 1998 E.J. Friedman Hill and the Sandia Corporation
Jess> date_time
cfg done
[day, month, simpleTime]
[300{hour=day, day=month, childname=simpleTime},
  300{hour=day, day=month, year=simpleTime},
... skipped 116 Assumptions
  300{childname=simpleTime, month=month, minute=day},
  300{month=month, year=simpleTime, minute=day}]
year=&
year=&month=&
leftover: year=&month=&minute=&
params: childname=12:57&day=07&hour=22&year=&month=&minute=&
pushdownUsingcurrent minute null 300{hour=day, day=month, childname=simpleTime}
minute not found 1
pushdownUsingcurrent hour null 300{hour=day, day=month, childname=simpleTime}
hour not found1
pushdownUsingcurrent simpleTime childname 300{hour=day, day=month, childname=simpleTime}
pushdown from 1 to 120
simpleTime not found1
day found
month found
resultevaluator: leftof 22 7
switching to evaluate
retracted month_found
retracted day_found
switching to check_config
childname=&
childname=&month=&
leftover: childname=&month=&minute=&
params: year=12:57&day=07&hour=22&childname=&month=&minute=&
pushdownUsingcurrent minute null 300{hour=day, day=month, year=simpleTime}
minute not found 2
pushdownUsingcurrent hour null 300{hour=day, day=month, year=simpleTime}
hour not found2
pushdownUsingcurrent simpleTime year 300{hour=day, day=month, year=simpleTime}
pushdown from 2 to 101
simpleTime not found2
```

```
day found
month found
resultevaluator: leftof 22 7
switching to evaluate
retracted month_found
retracted day_found
switching to check_config
childname=&
childname=&year=&
leftover: childname=&year=&minute=&
params: month=12:57&day=07&hour=22&childname=&year=&minute=&
pushdownUsingcurrent minute null 300{hour=day, day=month, month=simpleTime}
minute not found 3
pushdownUsingcurrent hour null 300{hour=day, day=month, month=simpleTime}
hour not found3
pushdownUsingcurrent simpleTime month 300{hour=day, day=month, month=simpleTime}
pushdown from 3 to 82
simpleTime not found3
day found
month found
resultevaluator: leftof 22 7
switching to evaluate
retracted month_found
retracted day_found
switching to check_config
childname=&
childname=&year=&
leftover: childname=&year=&month=&
params: minute=12:57&day=07&hour=22&childname=&year=&month=&
pushdownUsingcurrent minute null 300{hour=day, day=month, minute=simpleTime}
minute not found 4
pushdownUsingcurrent hour null 300{hour=day, day=month, minute=simpleTime}
hour not found4
pushdownUsingcurrent simpleTime minute 300{hour=day, day=month, minute=simpleTime}
pushdown from 4 to 63
simpleTime not found4
day found
month found
resultevaluator: leftof 22 7
switching to evaluate
retracted month_found
retracted day_found
switching to check_config
year=&
year=&month=&
leftover: year=&month=&minute=&
params: day=12:57&childname=07&hour=22&year=&month=&minute=&
pushdownUsingcurrent minute null 300{hour=day, day=simpleTime, childname=month}
minute not found 5
pushdownUsingcurrent month childname 300{hour=day, day=simpleTime, childname=month}
pushdown from 5 to 44
month not found 5
pushdownUsingcurrent hour null 300{hour=day, day=simpleTime, childname=month}
hour not found5
pushdownUsingcurrent simpleTime day 300{hour=day, day=simpleTime, childname=month}
pushdown from 5 to 37
simpleTime not found5
day found
switching to evaluate
retracted day_found
switching to check_config
```

```
childname=&
childname=&month=&
leftover: childname=&month=&minute=&
params: hour=12:57&year=07&day=22&childname=&month=&minute=&
pushdownUsingcurrent minute null 300{day=day, hour=simpleTime, year=month}
minute not found 6
pushdownUsingcurrent month year 300{day=day, hour=simpleTime, year=month}
pushdown from 6 to 21
month not found 6
pushdownUsingcurrent hour null 300{day=day, hour=simpleTime, year=month}
hour not found6
pushdownUsingcurrent simpleTime hour 300{day=day, hour=simpleTime, year=month}
pushdown from 6 to 18
simpleTime not found6
day found
switching to evaluate
retracted day_found
switching to check_config
switching to select_next_config
[day, month, minute, hour]
[400{hour=day, day=month, childname=minute, year=hour},
400{hour=day, day=month, childname=minute, month=hour},
400{hour=day, day=month, childname=minute, minute=hour},
...<354 intermediate combinations are left out>
400{hour=hour, month=month, year=minute, minute=day},
400{day=hour, month=month, year=minute, minute=day},
400{childname=hour, month=month, year=minute, minute=day}]
month=&
leftover: month=&minute=&
params: year=15&childname=57&day=07&hour=22&month=&minute=&
pushdownUsingcurrent minute childname 400{hour=day, day=month, childname=minute, year=hour}
pushdown from 1 to 360
minute not found 1
pushdownUsingcurrent hour year 400{hour=day, day=month, childname=minute, year=hour}
pushdown from 1 to 301
hour not found1
pushdownUsingcurrent simpleTime null 400{hour=day, day=month, childname=minute, year=hour}
simpleTime not found1
day found
month found
resultevaluator: leftof 22 7
switching to evaluate
retracted month_found
retracted day_found
switching to check_config
month=&
leftover: month=&minute=&
params: childname=15&year=57&day=07&hour=22&month=&minute=&
pushdownUsingcurrent minute year 400{hour=day, day=month, childname=hour, year=minute}
pushdown from 2 to 253
minute not found 2
pushdownUsingcurrent hour childname 400{hour=day, day=month, childname=hour, year=minute}
pushdown from 2 to 194
hour not found2
pushdownUsingcurrent simpleTime null 400{hour=day, day=month, childname=hour, year=minute}
simpleTime not found2
day found
month found
resultevaluator: leftof 22 7
switching to evaluate
retracted month_found
```

```
retracted day_found
switching to check_config
childname=&
leftover: childname=&year=&
params: minute=15&month=57&day=07&hour=22&childname=&year=&
pushdownUsingcurrent minute month 400{hour=day, day=month, month=minute, minute=hour}
pushdown from 3 to 146
minute not found 3
pushdownUsingcurrent simpleTime null 400{hour=day, day=month, month=minute, minute=hour}
simpleTime not found3
day found
month found
hour found
resultevaluator: leftof 22 7
switching to evaluate
retracted hour_found
retracted month_found
retracted day_found
switching to check_config
childname=&
leftover: childname=&year=&
params: month=15&minute=57&day=07&hour=22&childname=&year=&
pushdownUsingcurrent hour month 400{hour=day, day=month, month=hour, minute=minute}
pushdown from 4 to 111
hour not found4
pushdownUsingcurrent simpleTime null 400{hour=day, day=month, month=hour, minute=minute}
simpleTime not found4
day found
month found
minute found
resultevaluator: leftof 22 7
matched left: 22
matched right: 7
switching to evaluate
datum found
retracted minute_found
retracted month_found
retracted day_found
switching to check_config
year=&
leftover: year=&month=&
params: minute=15&day=57&childname=07&hour=22&year=&month=&
pushdownUsingcurrent minute day 400{hour=day, day=minute, childname=month, minute=hour}
pushdown from 5 to 76
minute not found 5
pushdownUsingcurrent month childname 400{hour=day, day=minute, childname=month, minute=hour}
pushdown from 5 to 53
month not found 5
pushdownUsingcurrent simpleTime null 400{hour=day, day=minute, childname=month, minute=hour}
simpleTime not found5
day found
hour found
switching to evaluate
retracted hour_found
retracted day_found
switching to check_config
childname=&
leftover: childname=&month=&
params: day=15&minute=57&year=07&hour=22&childname=&month=&
pushdownUsingcurrent simpleTime null 400{hour=day, day=hour, year=month, minute=minute}
simpleTime not found6
```

```
day found
month found
minute found
hour found
resultevaluator: leftof 15 57
resultevaluator: leftof 22 7
matched left: 22
matched right: 7
switching to evaluate
datum found
retracted hour_found
retracted minute_found
retracted month_found
retracted day_found
switching to check_config
childname=&
leftover: childname=&year=&
params: day=15&minute=57&month=07&hour=22&childname=&year=&
pushdownUsingcurrent simpleTime null 400{hour=day, day=hour, month=month, minute=minute}
simpleTime not found7
day found
month found
minute found
hour found
resultevaluator: leftof 15 57
resultevaluator: leftof 22 7
matched left: 22
matched right: 7
switching to evaluate
datum found
retracted hour_found
retracted minute_found
retracted month_found
retracted day_found
switching to check_config
childname=&
leftover: childname=&month=&
params: minute=15&hour=57&year=07&day=22&childname=&month=&
pushdownUsingcurrent minute hour 400{day=day, hour=minute, year=month, minute=hour}
pushdown from 8 to 41
minute not found 8
pushdownUsingcurrent month year 400{day=day, hour=minute, year=month, minute=hour}
pushdown from 8 to 24
month not found 8
pushdownUsingcurrent simpleTime null 400{day=day, hour=minute, year=month, minute=hour}
simpleTime not found8
day found
hour found
switching to evaluate
retracted hour_found
retracted day_found
switching to check_config
childname=&
leftover: childname=&year=&
params: hour=15&minute=57&month=07&day=22&childname=&year=&
pushdownUsingcurrent simpleTime null 400{day=day, hour=hour, month=month, minute=minute}
simpleTime not found9
day found
month found
minute found
hour found
```

```
resultevaluator: leftof 15 57
matched left: 15
matched right: 57
resultevaluator: leftof 22 7
matched left: 22
matched right: 7
switching to evaluate
time found
date found
ready
```

Bibliography

[1] Universal Interface Technologies Inc. Constantinos Phanouriou, *User interface markup language (uiml) draft specification*, http://www.uiml.org /specs/UIML2/specification.html, 2000.

[2] Geert de Haan, *Extended task-action grammar*, PhD dissertation, Eindhoven Technical University, ipo, 2000.

[3] Riley Giarratano, *Expert systems, principles and programming*, PWS-Kent Publishing, 1993.

[4] J. Hoppenbrouwers et al., *The grammalizer: A case tool based on textual analysis*, Submitted to the CAISE*97 Conference (1997).

[5] J. Hoppenbrouwers, B. van der Vos, and S. Hoppenbrouwers, *Nl structures and conceptual modelling: Grammalizing for kiss*, Special Issue of the Journal of Data and Knowledge Engineering (1997).

[6] A. Hussey and D. Carrington, *Using object-z to compare the mvc and pac architectures*, Proceedings of BCS-FACS Workshop: Formal Aspects of the Human Computer Interface (1996), 45–60.

[7] Gerald Kristen, *Kennis is macht*, tweede herziene uitgave van 'kiss-methode voor object orientatie' ed., Academic Service, 1998.

[8] Ian MacColl and David Carrington, *User interface correctness*, 2000.

[9] Pattie Maes, *Chi software agents tutorial*, http://pattie.www.media.mit.edu /people/pattie/CHI97/, 1997.

[10] Julie Meloni, *Php essentials*, Prima Publishing, 2000.

[11] H. Nwana, *Software agents: An overview*, The Knowledege Engineering Review (1996).

[12] XNS Public Trust Organization, *extensible name service*, http://www.xns.org, 2000.

[13] P. Palanque and R. Bastide, *Petri net based design of user-driven interfaces using the interactive cooperative objects formalism*, (1994), 383–300.

[14] M. Wooldridge and N. Jennings, *Intelligent agents: Theory and practice*, The Knowledege Engineering Review (1995).

[15] S. W. L. Yip and D. J. Robson, *Graphical user interfaces validation: A problem analysis and a strategy to solution*, Proceedings of the Twenty-Fourth Annual Hawaii International Conference on System Sciences (1991), 91–101.