U-WISH
Specification Techniques for Multi-Modal Dialogues

Olaf Donk, Betsy van Dijk, Anton Nijholt
{donk|bvdijk|anijholt}@cs.utwente.nl

July 6, 1999

Contents

Abstract

In this report we describe the development of a specification technique for specifying interactive web-based services. We wanted to design a language that can be a means of communication between designers and developers of interactive services, that makes it easier to develop web-based services fitted to the users and that shortens the pathway from design to implementation. The language, still under development, is based on process algebra and can be connected to the results of task analysis. We have been working on the automatic generation of executable prototypes out of the specifications. In this way the specification language can establish a connection between users, design and implementation. A first version of this language is available as well as prototype tools for executing the specifications. Ideas will be given as to how to make the connection between specifications and task analysis.

1 Introduction

This document is part of the deliverables of work-package (WP) 1.3 of the U-wish project as described in [NKD+98]. Here we give an overview of the research done for WP1.3, the results and the future plans. The goal of this workpackage of the U-wish project is to develop a specification technique that is not only suited for describing interactive web-based services and the interactions between users and the systems, but that can also serve as a means of communication between designers and developers of such systems. A language that is suited for this use has not been developed before, we are planning to base it on existing specification languages as much as possible.

In section 2 we discuss the focus of WP1.3, this section is recommended for everyone interested in the topics covered by or related to this project. Those readers interested mainly in our results and future plans are referred directly to the conclusions and section 8, in which we evaluate the specification language and we give suggestions for development of this language in WP2.3.

Readers interested in the resulting specification language may start by reading section 3, in which we give an overview of some existing specification techniques in order to obtain a starting point for the development of the new specification language. We discuss in section 4 several properties of specification languages in general with respect to our special demands and wishes. In the following section we will discuss the capacities and limitations of the techniques discussed in section 3 based on these requirements. The technical details of the specification language under development are covered in section 6, interested readers may pay special attention to this section, others may well skip this part. The actual use of the specification language under development is illustrated thoroughly by the specifications of the three information and transaction systems under study in the U-wish project. These specifications are explained in detail in section 7. For those interested we give the complete specifications of the three information systems under study in WP1.3 in the appendix.

2 Field of Interest

According to the description of the U-wish project WP1.3 focuses on

> the details of specific aspects of interaction in Web-based information and transaction environments, i.e. the effects of different interaction

styles, and possible schemes for allocating and integrating modalities in multi-modal dialogues.

First we will discuss this topic in order to describe our field of research in more detail. In literature the notion of modality is not always defined precisely. An interesting way to give a definition to modality and correspondingly to multi-modality can be found in [Ber94]. Bernsen proposes a categorisation of modalities based on five independent dimensions: *linguistic/non-linguistic, analogue/non-analogue, arbitrary/non- arbitrary, static/dynamic* and *media of expression*. After elimination meaningless locations in the thus created modality-space, this results in 28 different modalities. For example a text in Braille is linear, non-analogue, non-arbitrary, dynamical and its means of expression is haptic. As far as we know this categorisation of modalities is not used in the actual development of systems.

Another approach is to define modality as a communication channel. In this way speech, gestures, typed language and so on are different modalities. In systems using modalities a multi-modal dialogue may be created by using these different channels simultaneously. Such an approach is used to create a multi-modal environment in the DIVE- system [CH93], developed at SICS. A multi-modal dialogue is created by resolving anaphores in the natural language used by taking into account mouse-clicks.

In some systems different windows on the screen are also considered different modalities. The VMC [Nij99] and the AlFresco-system [ST93] are examples of systems using these notions. In the latter system ambiguities in the natural language used are resolved by using a mouse in a graphical 2D-environment. The VMC environment uses different windows to present information to the user, but it is not yet capable of combining input given in different windows.

In our development of a specification language capable of describing multi-modal dialogues we will consider both different communication channels and different windows as being different modalities. The systems under study in the U-wish project – the TNO-Human Factors website, the OLE2000 information system and the Virtual Music Centre of Twente University – focus on the use of typed natural language and graphical interfaces in combination with hypertext and virtual environments. Of course we do not want to limit ourselves to these combinations when the expressive power of our specification language is concerned.

The three systems of the U-wish project are all web-based, in the sense they are all developed in order to be accessible by the internet. All are executed from a web-browser, although none of the systems under study works equally well with both Microsoft Internet Explorer and Netscape Communicator. The OLE2000 works using Explorer, the TNO-site and the VMC only work when Netscape is used. An important aspect of the internet is the inter-connectivity of the systems that internet is made up of. Therefore all three systems (may) contain hyperlinks to other parts of the web. This is an important part of the functionality of a web-based system and we will explicitly describe the behaviour of the systems on this point.

In our research we do not claim explicitly that our specification language is suitable for web-based applications. The reason why we do not make explicit claims of our specification language being suitable for web-based applications is that theoretically all interactive computer systems are suitable for the internet. Theoretically, because at this moment there are limitations on the possibilities of the use of the internet. Most of these limitations are due to the technology currently in use. As we expect that most of these limitations will disappear as the technology evolves, we do not want to limit ourselves to the current possibilities.

As mentioned in the introduction we are developing a specification language that is suited for describing interactive systems, but can also be used for communicating between designers and programmers. Our goal in this project is to shorten the pathway between design and application. So on the one hand designers can see the results and consequences of their ideas easily and experiment with prototypes of their systems. On the other hand programmers have easy access to clear and unambiguous descriptions of the system that has to be implemented.

3 Known Specification Techniques

In the following section we'll give a short description of a number of specification languages. We'll introduce: natural language, GOMS, GTA, HDDL, Z and CSP. For more thorough descriptions the reader is referred to the references given in the corresponding subsections. In section 4 we will describe the criteria we've used to select the appropriate language for the development of our specification technique.

3.1 Natural language

First of all, natural language is not a formal method in itself. Its use lacks the precise and unambiguous communication that formal languages provide. The advantage of natural language is its enormous expressive and adaptive power. It is used often together with formal languages in order to explain the formal descriptions as formal languages on their own tend to be difficult to understand. Furthermore natural language is used to clarify the meaning of more formal specifications.

3.2 GOMS

The GOMS technique (Goals, Operators, Methods, and Selection Rules) [CMN83] is well suited for describing interactions. The GOMS method generates an analysis of the goals and sub-goals someone wants to achieve. The goals are ordered in a tree-like hierarchy. The operators in the GOMS description describe the actions the user of a system can perform. The methods are appropriate combinations of goals and operators, well-known to the user. Finally the selection rules describe how the user chooses between the various available methods at any time.

A GOMS specification, however, doesn't describe the interacting systems. The technique describes tasks with respect to systems and organisations. It is possible to use GOMS to generate descriptions of human performance, in advance based on specifications of the system the operator interacts with. GOMS may also be used to evaluate interactive systems. Several properties of the analysed system described can be derived, for example execution time and errors made. A number of variants of GOMS have been developed and choice between them should be based on what properties of the system specified are to be described.

3.3 GTA

The GTA method (Groupware Task Analysis) [vdVLB95] describes how a community of people works. The roles that various members of the community fulfill are described as well as how people cooperate. The actual work that is done is described by hierarchical task-description. The various objects that are relevant and the relation these objects have to one another is described in an object-structure. The resulting specification takes also that part of the system the user deals with into account, i.e. the user virtual machine (UVM). The UVM determines the semantics of the atomic tasks. These work- and object-structure serve in the GTA method as a basis for ETAG descriptions of the task-analysis. ETAG (Extended Task Action Grammar) [Tau90] is a declarative language in which tasks are described by decomposing them into hierarchical structured atomic actions.

3.4 HDDL

The HDDL language [Phi97a] [Phi97b] is developed by Philips for describing dialogues in automatic inquiry systems, involving speech recognition. It is a modular language, dialogues are decomposed into sub-dialogues and it incorporates a satisfying strategy for handling dialogue flow management. This strategy is explained in [AO95]. In use in real applications HDDL has proven to be a powerful language.

3.5 Z

The Z language, see [WD96] for a tutorial, is a formal specification technique based on logics. Specifications in Z are well suited to described structures, objects and relations between them. Also constraints hereupon can be described in Z. The language is based on logics and mathematics and uses the corresponding notation style. It is possible to specify dynamic aspects of systems using Z, but these do not combine with Z naturally.

3.6 CSP

The CSP algebra (Communicating Sequential Processes) developed by C.A.R. Hoare [Hoa85] is a kind of process algebra. In CSP concurrent processes and their interaction can be described. CSP is very well suited to describe multiple simultaneously active processes. It is also possible to assert certain properties of the systems specified. For example it is possible to check whether the interactive system cannot deadlock. Deadlock is a situation in which the system blocks every action.

4 Choice of Specification Technique

In this section we will describe a number of criteria on which we will score known specification techniques. These criteria also serve as guidelines in the development of the new specification language. In the next section we will evaluate the specification languages described in section 3.

Ease-of-use As we want to create a useful specification language, we want this language to be easy to use. Difficult interaction processes that involve a lot of actions may still be difficult to write down. But it should be possible to use our specification language to 'sketch' interactive systems.

Readability Specifications should serve as a medium of communication. The people that use them, therefore should be able to read them. As the specifications in the U-wish project should facilitate communication between web-designers and web-programmers special care should be taken on this point.

modularity The use of modular specifications enables reuse of parts of the specification in other systems. If these modules can be structured in an hierarchical architecture, it is possible to specify both the details of a systems as well as the more abstract general behaviour.

Possibility of execution If tools exist or can be made for executing specifications, it may be possible to use the specifications for prototyping.

Identification of tasks If it is possible to identify tasks and actions in the specification explicitly, it may be possible to develop specifications based on task-models. Especially in the U-wish project this is an interesting property. This may be a key-factor in the connection between WP1.2 and WP1.3

Expressive power Some properties are more easily described in a certain specification language and other properties more easily in another language. It is important for the usability of a specification language that one is using the proper language, i.e. it should be possible to write the important characteristics down straightforwardly.

Possibilities further development It is unlikely that a specification language exists that is 100% suitable for our use. Therefore we are looking for a language that can be used as a basis for further research and that can be expanded. In order to be able to expand a language we must known at least its syntax and semantics.

5 Capacities of Known Techniques

We will now discuss the advantages and drawbacks of the languages described in section 3. The criteria discussed above serve as guideline.

5.1 Natural Language

As natural language is not a formal method, but due to its adaptive capabilities it is useful for describing various systems. It is easy to use, but natural language tends to be highly ambiguous and it lacks expressive power when precise descriptions are needed. Reducing this ambiguity usually ends up in descriptions that are very difficult to read. Furthermore natural language does not have the merits of 'real' formal languages. It is not possible to derive properties of systems described solely in natural language, and it is not possible (yet) to generate executable prototypes automatically out of natural language specifications. We will use natural language however combined with a formal language for a first sketch of systems.

5.2 GOMS and GTA

GOMS and GTA are formal descriptions of people, or groups of people, and the tasks they perform. These descriptions can range from the description of the use of a simple calculator to the description of the daily work of a team of employees doing. The identification of tasks of the user of a system can be easily identified. Both GTA and GOMS, however, do not describe the systems people are working with. They describe the UVM at most, i.e. the systems as it appears to the user but not what is inside the system. These descriptions can not be made executable as the systems that should execute are not specified. Maybe it is possible to add this to the specifications, but then again a language is needed to expand these specifications. In section 8 we'll discuss future use of GTA and/or GOMS.

5.3 HDDL

Specifications of systems in HDDL are modular as the language demands a modular structure. HDDL is not really a description language, it is a programming language for dialogue systems, using its own management strategies. As HDDL is a programming language for dialogue systems it is not easy to identify the various tasks a user of the system performs. HDDL is tailored to handle questions, answer and other spoken utterances, but not mouse-clicks, navigation in virtual worlds, use of agents and so on. Furthermore, it is developed as it is and can not easily be adapted to future needs.

5.4 Z

The Z-language is used for specifications in industrial environments. It is suited to make modular specifications with different levels of abstraction. A drawback of Z for our purpose is its notation. The abundant use of mathematical symbols makes Z difficult to read and a large variety of symbols is needed. Actually one does not need all these symbols as a lot of them are shortcut notations, but Z demands very precise notations. So reducing the set of symbols used will not make the specifying easier. The most important drawback of Z is the static nature of what is described. It is difficult to describe changing environments and interaction between the system and users. This makes it also difficult to generate executable code out of these specifications.

5.5 CSP

In contrast to Z, CSP is based on actions and interactive processes. From a CSP-specification it is therefore quite simple to identify the task the user performs. As process algebra is based on actions its use may also enable us to make the connection between the results of hierarchical task analysis and the specifications of actual systems. We've mentioned the merits of this connection before in section 4.

It is possible to group parts of a CSP specification into modules and hide details to the outside. In this way a CSP specification may be read at different levels of abstraction. A number of executable languages are developed based on CSP and process algebra, for example LOTOS (Language Of Temporal Ordering Specifications) [ISO89]. The possibilities to make tools for executing seems promising and for WP1.3 we have made prototypes of tools for this purpose that are still under development. We have decided to develop a specification language based on CSP, in the next section we will discuss this in more detail.

6 New Specification Language

In a typical interactive system a number of processes are running at the same time and the user may choose to communicate with some of them. Generally these processes can be described by state transition diagrams. These diagrams are suitable to describe the possible actions of a process at a certain time as well as the communication a process is capable of. Although most processes in interactive systems can be described by state transition systems, the use of these diagrams is limited. For example it is not possible to create new instances of objects in a system that is totally described by state-based diagrams. In an actual application a number of processes is running simultaneously and the synchronisation of these processes is an important aspect when considering the exact behaviour of the system.

We have chosen to develop a specification language based on process algebra. This gives good insight in all possible actions at any moment and multi-modal dialogues can be handled. A process can handle input and output on different channels by being ready to perform different actions. Multi-modal communication by using multiple windows can be described by running multiple processes simultaneously.

The process algebra we used is a somewhat altered version of Communicating Sequential Processes (CSP) [Hoa85] With respect to standard CSP we have made some minor syntactic modifications to the language to make the typing easier. For example we have replaced the choice operator $[]$ by `[]` and the arrow $\rightarrow$ by `->`. For a more thorough description of the syntaxis and semantics of the CSP language we use, see [vSDZ99].

As process algebra does not put any claims on the actual results of the actions that take place, the execution of specifications results in a trace of these actions, but not in any behaviour as such. We wanted to investigate the apparent behaviour of the systems we have specified. Therefore we wanted to connect the specification and the actual appearance of the systems. The execution of the specification would then result in a running model of the system. The reasons to do this are threefold. First this connection will give us insight into the performance and limitations of the use of our specification language. Second it's easier to check the correspondence between the specifications we have made of the three systems and the actual systems. Finally, the use of this approach will enable us to use our language for prototyping purposes, possibly in the near future.

The demands on such a connection are rather severe, the mapping between the user interface (UI) and the processes of the specification should be a so-called retrieve function $\rho : UI \rightarrow CSP$ [ZCdR92]. This function should be total and surjective. This means that there is a CSP state corresponding to each state of the UI, and also that each possible CSP state is possible in the UI. If insufficient care is taken in creating this connection the behaviour of the created interface can be different from the specification. In that case the interface won't be useful for investigating the correctness of the specification.

In order to establish this connection we include several directives in the specification. These directives are preceding each process declaration for which there is a corresponding process in the UI. In the specifications we are working on at this moment the directives communicate with a TCL/TK process that controls the interface. TCL is an interactive scripting language that can cooperate with the GUI toolkit, TK [Ous90]. In the variant of the language to be developed in the second stage of the project, we do not necessarily have to use TCL/TK for this purpose. It is not evident that all important characteristics of a user interface that we want to express in our specifications can be programmed in TCL/TK. We believe however the connection as such is very useful in order to increase the usability of our specification language. The behaviour of the execution of the interface may serve as a dynamical documentation of the specification and hence increase the readability of the specification.

In [vSDZ99] we describe a specification of the VMC in which we have used these TCL/TK directives and we give an example of the possible looks of the resulting executable specifications.

7 Specifications of the Three Systems

In this section we will describe how we have developed the specifications of the three systems. The complete specifications will be given in appendix A, although we will explain most parts in the following sections. Off course the specifications do not contain all details of the systems specified. We have included those aspects of the systems that make up the general outline. Those parts of the systems that actually interact with the user or are meant to help the user to orientate within the systems are specified in more detail. Generally this means that we have not specified multiple instances of similar objects. For example we have not specified the different web-pages of the TNO-site, as they differ in content, but not in general functionality.

7.1 Natural Language

We will start with natural language descriptions to sketch important properties and behaviour of the systems. These descriptions serve as a starting point for the final, formal specifications. The formal specifications, given in section 7.2, are not final. This is because the insight gained during the development of these specifications will serve in the second stage of the project as guideline to further develop our specification language. During the development of this language we will also refine the specifications.

7.1.1 TNO-site

The TNO-website contains a number of pages, most of which are made up out of two different frames. The lower, large frame contains the text of the current page including hyperlinks, and the frame on top displays the location of the current page in the tree-like structure of the pages of the site. The indicator used to describe this location has the appearance of a traditional directory locator well known from for example URL's. The different subdirectories of this indicator are hyperlinks. These can be used to jump to pages one or more levels higher in the apparent hierarchy by a single click.

The higher frame also contains a search button which gives access to the search page. This page contains an entry field for a keyword and contains a link to a list of keywords. The user of the site may fill in one or more words in the entry field, the results of the search are displayed in a new page. On this page there are hyperlinks for each page of the site found and again a link to the list of keywords.

Unlike the other pages the list of keywords is made up of three different frames. Besides the two frames that make up all other pages a third frame is placed in between them. This third frame contains the alphabet and each letter is an anchor for the list of keywords in the lower frame. The use of this third frame keeps this list of anchors visible when the user scrolls or jumps to the lower parts of the list of keywords.

7.1.2 OLE2000

The OLE2000 information service is a web application consisting of a number of pages that are accessible by a series of buttons located in a frame at the left side of each page. One of these pages, accessible by the button 'nieuwe vraag stellen' (Dutch for 'ask new question'), contains a tool that can be used to compose a request for information out of a set of keywords. Once a keyword is chosen a number of new keywords becomes visible and these new keywords can be used to refine the request. This process repeats itself two to four times, and once the user has chosen the final keyword(s) the page that answers the composed question becomes visible. This page may contain options for more information that basically are hyperlinks to other pages.

7.1.3 THIS and the VMC

The VMC is a virtual environment that contains a number of interactive components. On the walls are posters that serve as hyperlinks to information about theater performances. If necessary a browser is started when a poster is clicked. There is a bookshelf on which the books behave in a similar manner as the posters. Doors can be opened and closed again, the user may make music on a keyboard and so on. A special functionality is embodied by an information board displaying a map of the theater. This map is a teleport and clicking it will transport the user to the corresponding place in the building.

Inside the VMC there is also an information desk. The user may ask here for information about theater performances and he/she may make reservations for these performances. The application that handles the information desk is called THIS (THeater Information System). In the current implementation of the theater the THIS system is also accessible when the user is not near the information desk. There is currently no interaction between the VMC and the THIS system.

THIS is an application that communicates with the user using written natural language (Dutch). The user can ask about specific performances or about the scheduled performances during a certain period of time. THIS will answer by using natural language or by filling in a table. The user may ask new questions by clicking in the table or by typing new questions. Whenever the user wants to make a reservation the system will ask for the necessary information, like number of persons and possible discount, that is not available to the system from the previous conversation. Finally there is a button to restart the whole system.

7.2 CSP

In the following sections we will describe the CSP specifications made of the three systems. The natural language descriptions given above will serve as a backbone. The descriptions given will be in the simple CSP variant. In sections 8 and 8 we will evaluate the capacities and limitations of the used CSP language.

7.2.1 TNO-site

The TNO-website contains two frames that interact with the user. The first frame, called the contents-frame from now on, contains the contents of the web-page that is currently in use, the second frame, the header- frame, contains a search-button and a directory-like location of the page in the tree-structure of the whole web-site. The interaction takes place by using the mouse. The frames and the mouse are modelled by the processes `Header_start`, `Contents_start` and `Mouse` respectively.

```
Tnosite where
Tnosite =    Header_start
          || Contents_start
          || Mouse,
```

The `Contents_start` process displays the contents of the main page of the web site and performs the action `first_display_header`. This latter action is used to synchronise the `Header_start`-process. In future extension of this specification these actions can be used to evoke TCL/TK-code. The `Contents_start`-process continues with the `Contents-` process. This process can react in three different ways. First it can respond to a mouse-click, `click_link`, on a hyperlink. In this situation the actions `display_normal_page` and `display_header` refresh the header- and contents-frame and the process repeats itself. Second it can respond indirectly to a mouse-click in the header, `click_header_link`, in which case the `Headerline`-process described below, performs the action `hdr_display_normal_page`. As a result of this action the content of the contents-frame is refreshed. The process continues by refreshing the header-frame and repeating itself. Finally the process can respond indirectly to a mouse-click on the search-button. The behaviour of the process is similar to the behaviour if a link in the header is clicked. Now the search-page is displayed in the contents-frame and the process continues not by repeating itself but by starting the `Search`-process.

```
Contents_start = (first_display_normal_page ->
                   first_display_header ->
                   Contents),

Contents = [ (click_link ->
               display_normal_page ->
               display_header ->
               Contents)
            [](hdr_display_normal_page ->
               display_header ->
               Contents)
            [](hdr_display_search_page ->
               display_header ->
               Search)],
```

The `Search`-process that handles the interactivity of the search-page includes the `Contents`-process. In addition to this the user may click on the link that opens the index-page ('Keyword List') or he/she may enter a keyword into the search_entry. The results of the search will be displayed by the action `display_search_result_page` and the process `Search_result` performs the corresponding interactivity. The results of the search are normal hyperlinks, handled by the included `Contents`-process, and the index-page is also accessible. When the `Index`-process is started the action `display_index_header` is also performed. It is used to display the alphabet of anchors displayed in a third frame between the normal header-frame and the contents-frame.

```
Search = [ Contents
         [](fill_search_entry ->
             display_search_result_page ->
             Search_result)
         [](click_index_link ->
             display_index_page ->
             display_header ->
             display_index_header ->
             Index)],

Search_result = [ Contents
                [](click_index_link ->
                    display_index_page ->
                    display_header ->
                    display_index_header ->
                    Index)],
```

The index-page, the list of keywords, is like a normal page but in addition to the interactivity of a normal page, handled by `Contents` it is possible to use the alphabet-anchors by clicking them, `click_index`.

```
Index = [ Contents
        [](click_index ->
            scroll_page ->
            Index)]
```

In order to monitor the actions performed by the user by using the mouse, we have modelled the `Mouse`-process that can perform a number of `click_...`-actions and then repeats itself. As this process is always ready for any action out of its alphabet it is merely useful for monitoring purposes. Furthermore its actions can be used to evoke TCL/TK-code.

```
Mouse = [ (click_link -> Mouse)
        [](click_index -> Mouse)
        [](click_index_link -> Mouse)
        [](click_search_button -> Mouse)
        [](click_header_link -> Mouse)],
```

The part of the specification that handles the interactivity of the header-frame consists of two processes: The process `Header_start` displays the header corresponding to the main-page of the site, by performing action `first_display_header`, and continues with the second process `Headerline` that repeats itself over and over. It can respond to mouse-clicks on the header and on the search-button. It can also synchronise with the action `display_header`.

```
Header_start = (first_display_header -> Headerline),

Headerline = [ (click_header_link ->
                 hdr_display_normal_page ->
                 display_header ->
                 Headerline)
             [](click_search_button ->
                 hdr_display_search_page ->
                 display_header ->
                 Headerline)
             [](display_header -> Headerline)]
```

7.2.2 OLE2000

The OLE2000 information service consists of two frames. The one on the left side contains a series of buttons that are used to access a number of pages that are viewed in the frame on the right side, the contents-frame . The process `Sidemenu_start` handles the behaviour of these buttons. The process that handles the behaviour of the page visible in the contents-frame is called `OleTwoK_contents`.

```
OleTwoK where
OleTwoK = Mouse
          || Sidemenu_start
          || OleTwoK_contents,
```

The `Sidemenu_start`-, and `Sidemenu`-process display a simple structure. The first process starts by performing the action `start_algemene_informatie` and then continues by running the second process. This second process performs an action `set_...` after the user performs, by means of the mouse, an `click_...`-action. Then the `Sidemenu`-process repeats itself.

```
Sidemenu_start = (start_algemene_informatie -> Sidemenu),

Sidemenu = [ (click_algemene_informatie ->
               set_algemene_informatie ->
               Sidemenu)
           [](click_actuele_informatie ->
               set_actuele_informatie ->
               Sidemenu)
           [](click_andere_informatie ->
               set_andere_informatie ->
               Sidemenu)
           [](click_nieuwe_vraag_stellen ->
               set_nieuwe_vraag ->
               Sidemenu)
           [](click_futuretrain ->
               set_futuretrain ->
               Sidemenu)],
```

The `Mouse`-process is modelled like in the TNO-site specification. It may perform a `click_...`-action and then it repeats itself. The pages in the contents-frame are changed by the actions `echo_...`. The `OleTwoK_contents`-process will restart when one of these actions is performed. An exception to this behaviour occurs when the button 'nieuwe vraag stellen' is clicked by the user. The process then continues with the three `..._question`-processes. Depending on the availability of more keywords to refine the request for information the `Expand_question`-process repeats itself or continues with the `Answer_question`-process. The user may also click one of the buttons in the frame on the left side and the system will respond by changing the contents of the right side frame. To enable this possibility the `OleTwoK_contents`-process is also accessible as an alternative to clicking more keywords or options.

```
OleTwoK_contents = [ (set_algemene_informatie ->
                       echo_algemene_informatie ->
                       OleTwoK_contents)
                   [](start_algemene_informatie ->
                       echo_algemene_informatie ->
                       OleTwoK_contents)
                   [](set_actuele_informatie ->
                       echo_actuele_informatie ->
                       OleTwoK_contents)
                   [](set_andere_informatie ->
                       echo_andere_informatie ->
                       OleTwoK_contents)
                   [](set_nieuwe_vraag ->
                       Compose_question)
                   [](set_futuretrain ->
                       echo_futuretrain ->
                       OleTwoK_contents)],
Compose_question = (echo_list ->
                     [ OleTwoK_contents
                     [](click_keyword ->
                         determine_expand_set ->
                         Expand_question)]),

Expand_question = (echo_list ->
                    [ OleTwoK_contents
                    [](click_keyword ->
                        [ (determine_expand_set ->
                           Expand_question)
                        [](determine_answer ->
                           Answer_question)])]),

Answer_question = (echo_answer ->
                    [ OleTwoK_contents
                    [](click_option ->
                        determine_answer ->
                        Answer_quest
```

7.2.3 THIS and the VMC

THIS

First we will specify the apparent behaviour of THIS when the user doesn't use the mouse. The user can only enter questions by entering them in the entry field. Also Karin cannot answer by using the table as in the THIS system. The systems consists of five parallel processes, one for each frame, one for the user and one representing Karin.

```
This where This = Entry || Dialogue || Karin || User || Status
```

The user can only enter utterances. Of course the user can watch the systems behaviour, but for obvious reasons this part of the user is not modelled.

```
User = (entryfield -> User)
```

The entry field accepts an utterance, passes it to the dialogue recorder and clears itself. Finally it passes the utterance to Karin. This process is repeated over and over.

```
Entry = (entryfield ->
          entrytodialogue ->
          clearentryfield ->
          passtokarin ->
          Entry)
```

The dialogue recorder accepts alternating utterances from the entry-field and Karin. It starts with waiting for an utterance from Karin, as she is starting the conversation ('How can I help you?'). The status-process just goes on and on accepting status messages. The actual displaying is not modelled in these processes, it is assumed that the displaying is a side-effect from the acceptance of a message. Modelling of the displaying would unnecessarily clutter the trace of the model.

```
Dialogue = ( karintodialogue ->
               entrytodialogue ->
               Dialogue),

Status = (passtostatus -> Status)
```

Karin consists of two parallel processes, Karinstart and Iq. Iq models the conversation module of Karin. Here it accepts an utterance and reacts by returning a number of status messages, possibly none, and finally returning an answer to the question.

```
Karin = Karinstart || Iq,

Iq = (passtoiq -> Iqreact),

Iqreact = [  (getfromiq -> Iq)
          [] (getstatus -> Iqreact)]
```

The process Karinstart starts a new conversation and makes Karin welcome the user. Once the conversation is started Karin accepts questions, passes them to her I.Q. and returns the reactions from `Iq` to the dialogue recorder.

```
Karinstart = (passtoiq ->
                getfromiq ->
                karintodialogue ->
                Karinbehaviour),

Karinbehaviour = (passtokarin ->
                    passtoiq ->
                    Waitreaction),

Waitreaction =  [ (getfromiq ->
                     karintodialogue ->
                     Karinbehaviour)
                []（getstatus ->
                     passtostatus ->
                     Waitreaction)]
```

Specification of the complete dialogue

The processes `Table`, `Mouse` and `Nieuw_button` are added to THIS resulting in the following process definition:

```
This = Entry
       || Dialogue
       || Status
       || Table
       || Karin
       || User
       || Mouse
       || Nieuw_button
```

As it is possible to ask questions by clicking in the table the process `Dialogue` is changed into:

```
Dialogue = (karintodialogue ->
              [ (entrytodialogue ->
                  Dialogue)
              [](questiontodialogue ->
                  Dialogue)])
```

The process Mouse doesn't care much, it is an abstract version of the actual program that interprets the mouse and manages the pointer and cursor. The action mouse_out takes place as the mouse is clicked outside the pop-up-menu containing the question about the selected performance.

```
 Mouse = [ (nieuw_button -> Mouse)
        [](click_table -> Mouse)
        [](mouse_out -> Mouse)
        [](ask_table -> Mouse)]
```

The process Emptytable manages the behaviour of the table before Karin has written information into it. Although it is of no use, in the current THIS it is possible to click an empty performance. The process Filledtable speaks to itself. The process Select_menu models the behaviour of the table when a performance (possibly an empty one) is selected and the pop-up-menu is visible. Finally the process Select models how the table behaves when the pop-up-menu has disappeared.

```
Table = Emptytable,

Emptytable = [ (fill_table -> Filledtable)
             [](new_table -> Emptytable)
             [](click_table -> Select_menu)],

Filledtable = [ (fill_table -> Filledtable)
              [](new_table -> Emptytable)
              [](click_table -> Select_menu )],

Select_menu = [ (mouse_out -> Select)
              [](ask_table -> question_out -> Select)],

Select = [ (fill_table -> Filledtable)
         [](new_table -> Emptytable)
         [](click_table -> Select_menu)]
```

The process associated with the button that starts a new dialogue, named 'Nieuw' in Dutch, simply waits untill it is clicked, then tries to reset Karin and restarts itself.

```
Nieuw_button = (nieuw_button ->
                resetkarin ->
                Nieuw_button)
```

Karin is much the same, she now can be reset and she accepts question from the pop-up-menu. She also can answer by filling the table. This will always be accompanied by a verbal response in the dialogue window. The Iq process is slightly altered to allow Karin to be reset.

```
Karinbehaviour = [ (resetkarin -> new_table -> Karinstart)
                 [](passtokarin ->
                      passtoiq ->
                      Waitreaction)
                 [](question_out ->
                      questiontodialogue ->
                      passtoiq ->
                      Waitreaction)],

Waitreaction = [ (getfromiq ->
                   [ (karintodialogue ->
                        Karinbehaviour)
                   [](fill_table ->
                        karintodialogue ->
                        Karinbehaviour)])
               [](getstatus ->
                    passtostatus ->
                    Waitreaction)],

Iq = [  (passtoiq -> Iqreact)
     [] (startiq -> Iqreact)],
```

The VMC

A subset of the interactive world of the VMC is modelled. The world, Vmc, consists of a number of parallel processes. The This-process is the Karin-system described in the previous section. The This_manager is used to activate and deactivate the processes under This by means of the actions this_on and this_off. The small changes made to these processes are self-explanatory and can be found in the appendix. So:

```
Vmc = Mouse
      || Avatar
      || World
      || Bookshelf
      || Poster
      || Door
      || Map
      || Browser_manager
      || This_manager
      || This
```

The processes that model the behaviour of books and posters evoke the Browser_manager to start up a new internet-browser. The door can be opened and closed again...

```
Bookshelf = (book_click -> evoke_browser_book -> Bookshelf),

Poster = (poster_click -> evoke_browser_poster -> Poster),

Browser_manager = [ (evoke_browser_book ->
                      Browser_manager)
                  [](evoke_browser_poster ->
                      Browser_manager)],
Door = (door_click ->
         open_door ->
         door_click ->
         close_door ->
         Door)
```

The processes for moving around are a bit more complex. The Map-process, that serves as a teleport in the VMC, communicates with the Avatar, that on its turn can respond to the map and to move-commands from the user given by the mouse. The Avatar-process checks with the world to see whether a move asked for by the mouse is possible or not

```
Map = (map_click -> goto_map -> Map),

Avatar = [ (try_move -> check_world ->
             [ (possible_move -> goto_step -> Avatar)
             [](not_possible_move -> Avatar)])
         [](goto_map -> Avatar)],

World = (check_world ->
          [ (possible_move -> World)
          [](not_possible_move -> World)])
```

The mouse that is used for most communication between user and system is modelled like this:

```
Mouse =  [ (book_click -> Mouse)
         [](door_click -> Mouse)
         [](map_click -> Mouse)
         [](poster_click -> Mouse)
         [](try_move -> Mouse)
         [](activate_this -> Mouse)
         [](deactivate_this -> Mouse)
         [](nieuw_button -> Mouse)
         [](click_table -> Mouse)
         [](mouse_out -> Mouse)
         [](ask_table -> Mouse)]
```

8 Evaluation and Future Plans

In this section we describe our experience with the language we have developed so far. We will compare this language again with the criteria mentioned in section 4. The first criterion is easy-of-use. Based on our own experiences using the language we think the language is easy to use and makes it possible to sketch interactive processes. We have found that with some experience it is also quite easy to specify interactive systems consisting of a number of simultaneously running components. However, one has to be very careful when specifying larger systems. As more processes have to cooperate one has to deal with the synchronisation properties of CSP more carefully. Also some experience is needed to avoid dead-lock situations, i.e. a kind of stalemate, see [Hoa85]. Sometimes this can only be avoided by using clever naming conventions. It is important the specification language is easy to use and we will have to take care on this point in WP2.3. Experiences other people have when using the language are important as we ourselves are not fair when judging our own language.

A next criterion to judge the specification language upon is its readability. Like with the previous criterion, this is difficult to determine for us. We ourselves are used to our specification language, it will again be necessary to introduce others to our language in order to obtain a less subjective view. We will address the problem of readability in WP2.3.

The results of the developed specification language on the following criteria we can evaluate more objectively. At this stage of the development it seems likely that it is possible to structure the specifications very well. CSP has good capacities of structuring different levels of abstraction. We have plans for expanding the CSP language with facilities for creating modular specifications. The functionality basically needed exists already in CSP. On this moment we are also working on a way to execute CSP specifications. We are developing tools for automatically generating executable code based on the CSP specifications, possibly expanded with some code, possibly TCL/TK-code, for generating the interface. The results only preliminary yet, but our effort seems worthwhile and we are planning to do further research on this point in WP2.3.

Special care will be taken in WP2.3 on the 'identification-of-tasks'-criterion. The use of CSP offers an easy access to the actions the user of an interactive systems performs. And as we are interested in the development of specifications out of the results of task analysis, we will investigate the possibilities of this connection during the next year of the project. Concerning the last two criteria, 'expressive power' and 'possibilities further development', we expect no great problems in the nearby future. Currently we are investigating the possibilities of specifying CSP-processes that can be created during execution. This could improve the results of the specification language but is not absolutely necessary to make CSP useful as a basis for our language.

9 Conclusions

We have developed a first version of a specification language for interactive services. Our goal during the development has been to establish a connection between users, design and implementation. Therefore we have tried on the one hand to take the users as a starting point for our specification. This is done by connecting process algebra and task analysis. We expect that this connection will prove useful but we can not state this for certain on this moment. Therefore it will be necessary to further investigate this connection during the second year of the project. On the other hand we wanted our specification to be as precise and close to implementation as possible. The possibilities and limitations of the automatic generation of executable prototypes have not been investigated thoroughly yet, our result so far are preliminary on this point. During the second year of the project we have to investigate this thoroughly. To conclude this overview of the current status of our results, we want to state that we think that the combination of task analysis, process algebra and automatic generation of executable prototypes has given promising preliminary results and that further research on this point will be worthwhile.

Summarised we think that the research that is done so far and that we are planning to do during the next year can be powerful for bringing users and interactive services together. As our specifications in the developed language can be fitted onto the results of task analysis, the resulting interactive services are tailored to fit the users. Also the generated executable prototypes provide a way to evaluate the design of the services in an early stage of the development.

References

[AO95] Harald Aust and Martin Oerder, *Dialogue control in automatic inquiry systems*, Proceedings of the Ninth Twente Workshop on Language Technology (TWLT9) (J.A. Andernach, S.P. van de Burgt, and G.F. van der Hoeven, eds.), 1995, pp. 45–49.

[Ber94] Niels Ole Bernsen, *Foundations of multimodal representations. a taxonomy of representation modalities.*, Interacting With Computers **6** (1994), 347–371.

[CH93] Carlsson and Hagsand, *DIVE - a multi-user virtual reality system*, IEEE Virtual Reality Annual International Symposium (VRAIS), 1993, pp. 394–400.

[CMN83] S.K. Card, T.P. Moran, and A. Newell, *The psychology of human-computer interaction*, Hillsdale, NJ, Lawrence Erlbaum Associates, 1983.

[Hoa85] C.A.R. Hoare, *Communicating sequential processes*, Prentice-Hall International, 1985.

[ISO89] *Information processing systems – Open Systems Interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour*, 1989, ISO 8807:1989.

[Nij99] Anton Nijholt, *The Twente theatre environment: Agents and interactions*, Proceedings of the Fifteenth Twente Workshop on Language Technology (TWLT15) (A. Nijholt, O.A. Donk, and E.M.A.G. van Dijk, eds.), 1999, pp. 195–212.

[NKD$^+$98] M.A. Neerincx, J.H. Kerstholt, E.M.A.G. van Dijk, A. Nijholt, S. Pemberton, and G.P.J. Spenkelink, *Uwish – web-based services for information and commerce: User-centered design principles, methods and applications*, 1998, http://www.cwi.nl/projects/uwish/.

[Ous90] John Ousterhout, *Tcl: An embeddable command language*, Proceedings of USENIX Winter Conference, 1990.

[Phi97a] Philips Dialogue Systems, Aachen, Germany, *Speechmania 2.0: HDDL reference manual*, 1997.

[Phi97b] Philips Dialogue Systems, Aachen, Germany, *Speechmania 2.0: HDDL user's guide*, 1997.

[ST93] Oliviero Stock and Team, *ALFRESCO: Enjoying the combination of NLP and hypermedia for information exploration*, 1993.

[Tau90] M.J. Tauber, *ETAG: Extended Task Action Grammar - a language for the description of the user's task language*, Human-Computer Interaction – INTERACT '90 (D. Diaper et al., ed.), Elsevier Science Publishers B.V. (North-Holland), 1990.

[vdVLB95] G.C. van der Veer, B.F. Lenting, and B.A.J. Bergevoet, *Groupware taks analysis: Modeling complexity*, 1995.

[vSDZ99] Boris van Schooten, Olaf Donk, and Job Zwiers, *Modelling interaction in virtual environments using proces algebra*, Proceedings of the Fifteenth Twente Workshop on Language Technology (TWLT15) (A. Nijholt, O.A. Donk, and E.M.A.G. van Dijk, eds.), 1999, pp. 195–212.

[WD96] Jim Woodcock and Jim Davies, *Using Z - specification, refinement, and proof*, Prentice Hall Europe, 1996.

[ZCdR92] J. Zwiers, J. Coenen, and W. P. de Roever, *A note on compositional refinement*, Proceedings of the 5th refinement workshop, 1992, pp. 342–366.

A Complete Specifications

```
Tnosite where
Tnosite =    Header_start
          || Contents_start
          || Mouse,

 -- Contents_start starts both the header-frame on top of the page
 -- and the 'standard'-frame. After that it continues with the
 -- process Contents.

Contents_start = (first_display_normal_page ->
                   first_display_header ->
                   Contents),

 -- Contents manages the contents of the 'standard'-frame. It
 -- reacts on clicked links (click_link). It can also be refreshed
 -- by the actions hdr_display_normal_page and
 -- hdr_display_search_page, both synchronised by the process
 -- Headerline. If the standard-frame displays the search-page
 -- the process continues with Search.

Contents = [ (click_link ->
               display_normal_page ->
               display_header ->
               Contents)
           [](hdr_display_normal_page ->
               display_header ->
               Contents)
           [](hdr_display_search_page ->
               display_header ->
               Search)],

Search = [ Contents
         [](fill_search_entry ->
             display_search_result_page ->
             Search_result)
         [](click_index_link ->
             display_index_page ->
             display_index_header ->
             Index)],

Search_result = [ Contents
                []( Click_index_link ->
                display_index_page ->
               display_index_header ->
                    Index)],

 -- the list of keywords, the user may scroll through it clicking
 -- the index, i.e. the alphabet displayed between the standard-
 -- frame and the header.
```

```
Index = [ Contents
        [](click_index ->
            scroll_page ->
            Index)],

 -- an overview of all possible mouse-clicks

Mouse = [ (click_link -> Mouse)
        [](click_index -> Mouse)
        [](click_index_link -> Mouse)
        [](click_search_button -> Mouse)
        [](click_header_link -> Mouse)],

Header_start = (first_display_header -> Headerline),

Headerline = [ (click_header_link ->
                 hdr_display_normal_page ->
                 display_header ->
                 Headerline)
             [](click_search_button ->
                 hdr_display_search_page ->
                 display_header ->
                 Headerline)
             [](display_header -> Headerline)]
```

```
OleTwoK where
OleTwoK = Mouse
          || Sidemenu_start
          || OleTwoK_contents,

Mouse =  [ (click_algemene_informatie -> Mouse)
         [](click_actuele_informatie -> Mouse)
         [](click_andere_informatie -> Mouse)
         [](click_nieuwe_vraag_stellen -> Mouse)
         [](click_futuretrain -> Mouse)
         [](click_keyword -> Mouse)
         [](click_option -> Mouse)],

Sidemenu_start = (start_algemene_informatie -> Sidemenu),

Sidemenu = [ (click_algemene_informatie ->
               set_algemene_informatie ->
               Sidemenu)
           [](click_actuele_informatie ->
               set_actuele_informatie ->
               Sidemenu)
           [](click_andere_informatie ->
               set_andere_informatie ->
               Sidemenu)
           [](click_nieuwe_vraag_stellen ->
               set_nieuwe_vraag ->
               Sidemenu)
           [](click_futuretrain ->
               set_futuretrain ->
               Sidemenu)],

OleTwoK_contents = [ (set_algemene_informatie ->
                       echo_algemene_informatie ->
                       OleTwoK_contents)
                   [](start_algemene_informatie ->
                       echo_algemene_informatie ->
                       OleTwoK_contents)
                   [](set_actuele_informatie ->
                       echo_actuele_informatie ->
                       OleTwoK_contents)
                   [](set_andere_informatie ->
                       echo_andere_informatie ->
                       OleTwoK_contents)
                   [](set_nieuwe_vraag -> Compose_question)
                   [](set_futuretrain ->
                       echo_futuretrain ->
                       OleTwoK_contents)],

Compose_question = (echo_list ->
                     [ OleTwoK_contents
                     [](click_keyword -> determine_expand_set
                         -> Expand_question)]),

Expand_question = (echo_list ->
                     [ OleTwoK_contents
                     [](click_keyword ->
                       [ (determine_expand_set ->
                          Expand_question)
                       [](determine_answer ->
                          Answer_question)])]),

Answer_question = (echo_answer ->
                     [ OleTwoK_contents
                     [](click_option ->
                         determine_answer ->
                         Answer_question)])
```

```
Vmc where
Vmc = Mouse
      || Avatar
      || World
      || Bookshelf
      || Poster
      || Door
      || Map
      --- Browser_manager is modelled as one always present
      --- process, I'd rather model the browsers such that
      --- they can be generated during runtime.
      || Browser_manager
      || This_manager
      || This,

 --- Mouse is the process that interprets the actual mouse, the one
 --- with the wire and the buttons. This process should communicate
 --- with the world. For the moment this part is left out.

Mouse =  [ (book_click -> Mouse)
         [](door_click -> Mouse)
         [](map_click -> Mouse)
         [](poster_click -> Mouse)
         [](try_move -> Mouse)
         [](activate_this -> Mouse)
         [](deactivate_this -> Mouse)
         [](nieuw_button -> Mouse)
         [](click_table -> Mouse)
         [](mouse_out -> Mouse)
         [](ask_table -> Mouse)],

User = (entryfield -> User),

This_manager = (activate_this -> this_on ->
                  deactivate_this -> this_off
                  -> This_manager),

This = Entry
       || Dialogue
       || Status
       || Table
       || Karin
       || User
       || Nieuw_button,

Entry = (this_on -> Entry_active),

Entry_active = [ (entryfield ->
                   entrytodialogue ->
                   clearentryfield ->
                   passtokarin ->
                   Entry_active)
               [](this_off -> Entry)],
Dialogue = (karintodialogue ->
              [ (entrytodialogue -> Dialogue)
              [](questiontodialogue ->
                  Dialogue)]),

Status = (this_on -> Status_active),

Status_active = [ (passtostatus -> Status_active)
                []( this_off -> Status)],

--- Table processes ---

Table = (this_on -> Emptytable),

Emptytable = [ (fill_table -> Filledtable)
             [](new_table -> Emptytable)
             [](click_table -> Select_menu)
             [](this_off -> Table)],

Filledtable = [ (fill_table -> Filledtable)
              [](new_table -> Emptytable)
              [](click_table -> Select_menu )
              [](this_off -> Table)],

Select_menu = [ (mouse_out -> Select)
              [](ask_table -> question_out ->
              Select) [](this_off -> Table)],

Select = [ (fill_table -> Filledtable)
         [](new_table -> Emptytable)
         [](click_table -> Select_menu)
         [](this_off -> Table)],

--- the 'nieuw'-button can only be pressed when Karin is not busy

Nieuw_button = (this_on -> Nieuw_button_active),

Nieuw_button_active = [ (nieuw_button ->
                          resetkarin ->
                          Nieuw_button_active)
                      [](this_off -> Nieuw_button)],

Karin = Karinstart || Iq,

Karinstart = (this_on -> startiq ->
                getfromiq ->
                karintodialogue ->
                Karinbehaviour),

Karinbehaviour = [ (resetkarin -> new_table -> Karinstart)
                 [](this_off -> Karinstart)
                 [](passtokarin ->
                      passtoiq ->
                      Waitreaction)
                 [](question_out ->
                      questiontodialogue ->
                      passtoiq ->
                      Waitreaction)],

Waitreaction = [ (getfromiq ->
                    [ (karintodialogue ->
                        Karinbehaviour)
                    [](fill_table ->
                        karintodialogue ->
                        Karinbehaviour)])
                    [](getstatus ->
                        passtostatus ->
                        Waitreaction)],

Iq = [  (passtoiq -> Iqreact)
     [] (startiq -> Iqreact)],

Iqreact = [ (getfromiq -> Iq)
          [](getstatus -> Iqreact)],

 --- It would be better to pass book and poster as
 --- variable of evoke_browser. They should not
 --- synchronise

Bookshelf = (book_click -> evoke_browser_book -> Bookshelf),

Poster = (poster_click -> evoke_browser_poster -> Poster),

Browser_manager = [ (evoke_browser_book -> Browser_manager)
                  [](evoke_browser_poster -> Browser_manager)],

Door = (door_click ->
         open_door ->
         door_click ->
         close_door ->
         Door),

Map = (map_click -> goto_map -> Map),

 --- some parameter should be passed on to the move action,
 --- and accordingly to the check_world action

Avatar = [ (try_move -> check_world ->
             [ (possible_move -> goto_step -> Avatar)
             [](not_possible_move -> Avatar)])
             [](goto_map -> Avatar)],

World = (check_world ->
          [ (possible_move -> World)
          [](not_possible_move -> World)])
```