Modelling Interaction in Virtual Environments using Process Algebra

Boris van Schooten, Olaf Donk, Job Zwiers
University of Twente
P.O. Box 217, 7500 AE Enschede
{schooten|donk|zwiers}@cs.utwente.nl

Abstract

Virtual environments (VEs) are often a complex mixture of novel and traditional user interface strategies, and also incorporate real-time dynamics and parallelism. We describe a modelling technique we are developing, which is based on the process algebra CSP. It is shown how VE systems and tasks can be modelled in CSP, and how a prototype system can be generated from the system specification by mapping a subset of CSP signals onto user interface functionality.

Keywords: Interaction Design, Process Algebra, Prototyping, Multimodal Interaction, Agent-Oriented Modelling, Virtual Environments, Task Analysis

1 Introduction

When designing Virtual Environments (VEs) it seems natural to design them as consisting of multiple 'agents', 'objects', or processes. For example, MUD and Virtual Reality (VR) systems are usually built out of a large number of concurrent (i.e. parallel and communicating) processes. In our own VE project, the Virtual Music Centre (VMC) [34], a high degree of concurrency exists also. The agents within the VE, the user navigating through it, as well as some more traditional 'desktop' user interface elements operate or may be operated concurrently. Plans also exist to enable multiple users to enter the VMC simultaneously, but no precise model yet exists of what information is shared and communicated between the users.

However, process concurrency is known to be a difficult computer science problem. Typical concurrency problems are timing problems and unaccounted-for unusual situations, and may surface as anything ranging from crashing bugs to ill behaviour that may be classified as usability problems. This is already seen with systems modelled after the desktop metaphor. In some ways, the desktop model is a form of VE, and may be considered a relatively established and mature model. Desktop systems often have concurrency bugs. For example, multiple windows accidentally access the same data concurrently, or the effects of a resize button or scrollbar is not notified properly, and the system crashes or the display becomes inconsistent. Concurrency-related usability problems also occur. For example, some windows do not allow the user to concurrently access other windows because of limitations of the software, and freeze the rest of the application. In some cases, this is not clear to the user, who may even think the application crashed.

It is attractive to enhance VEs with multimodal interaction. Here, we define a modality as a distinct channel through which information may be conveyed, either from the computer to the user (output modalities: ears, eyes) or the other way (input modalities: microphone, keyboard). In the literature, multimodality is interpreted in different ways by different authors, depending on their point of view: modalities may be classified not just according to physical distinction criteria, but also to cognitive or technological ones [5] [32] [27]. So, it is also possible to think of a line of text and a graphic as different modalities, because they have different cognitive properties. Bernsen [5] classifies continuous and isolated-word speech as different modalities. Isolated-word speech exists only because of technological constraints, and has distinct technological and cognitive properties. In the same vein, it is even conceivable to view working with different windows as multimodality. For example, a pull-down menu and a text entry field are two distinct technological concepts, and have different cognitive properties as well. Here, the concept of multimodality touches

with the multi-agent concept as described above. This suggests that multimodality could naturally be modelled using concurrent processes, having a process for each modality.

1.1 Formal methods in HCI

As argued above, VE development seems to call for a modelling technique which enables designers to work easily and naturally with multiple concurrent processes. From a software engineering (SE) point of view, reducing the amount of concurrency may be more desirable. From a HCI point of view however, modelling systems as many parallel processes standing for interaction components may be closer to the conceptual or mental model the HCI designers have in mind, resulting in less problems when mapping the conceptual model to the implementation model [23]. It is also attractive to use a formal technique, because of its acclaimed properties [20]:

1. Conceptual clarity, aiding communication between the parties involved, in particular, the HCI and SE parties [14].
2. Enforcement of precision while designing, and stimulation of a specific way of thinking about the design. This may lead to timely consideration of issues that are otherwise overlooked.
3. Ability to manually or automatically verify usability-related properties. Various examples of such properties are found in the HCI literature. They can be separated into validity properties, such as reachability of states, deadlock-freeness, [27], complementarity, equivalence, and redundancy of modalities [10], and cognitive ones, such as task duration, cognitive load, and interface consistency [16].

Looking at the interactive systems literature, various attempts at formal specification have been made using various formal languages, ranging from those suitable for highly declarative and constraint-based descriptions (predicate logic, functional languages) to highly procedural ones (production systems, state automata, Petri nets, process algebras). See [20], [36], or [19] for examples of each. However, they remain little used in practice. This may in part be caused by factors other than their usefulness [7], but there are few clear results showing that they really work [15]. We will discuss some of the potential causes of problems with formal methods, and why formal methods may nevertheless be useful.

Sometimes it is claimed that an advantage of formal specification is its relative completeness with respect to reality [14], but exactly the opposite is the case. Aspects which may be important are necessarily omitted to make the specification tractable [20], or may be omitted unwittingly, even in simple cases [18]. Especially in (but not exclusively to) HCI, some of these aspects are even practically unspecifiable. Examples are the general context in which a system is used, and unanticipated user behaviour with a new system, evoked by unanticipated user knowledge and expectations. This means that, in general, some unspecified aspects remain that can only be examined by trial and error. In the case of HCI, this may also require repeated communication with the users. As a complement to developing by means of specifications, HCI often uses example-based techniques, such as mock-ups, scenarios, and prototypes. These are then considered by the developer(s) in detail, or shown to the end users [33]. However, example-based techniques may also lead to underemphasis of those aspects that can not be derived from a number of examples only, and should in turn be complemented with precise descriptions [3].

Next to this, the enforcement of precision is often seen as a drawback rather than an advantage. This may be caused by an inappropriate level of abstraction, but also by incompatibility of the specification with the developers' needs. It is important that the specification should be usable as the primary memory aid in various stages of the development process by various parties (display-based reasoning) [12]. Therefore, it should also conform to the nature of the design tasks, otherwise the specification may become a burden rather than an aid. In order to ensure that a method is useful as a thinking aid, we will have to look at what the different parties involved in the design of interactive systems need.

HCI specification techniques often document the user and the user's view. The user's task is usually documented using hierarchical task models, and the user's conceptual knowledge is usually documented using semantic nets supplemented with plain-text descriptions [28] [37]. The most popular formal HCI models are formal instances of such models, such as Goals Operators Methods Selection-rules (GOMS) [16] and Extended Task Action Grammar (ETAG) [14] (see [19] for an overview). Other models include flowchart models [22]. Note that, like many example-based methods, these formal models are mostly procedure-oriented rather than declara-

tive or constraint-oriented, describing what happens step by step. Some of the more detailed models are also capable of modelling tasks that can be executed simultaneously, though usually, the tasks may not be concurrent, e.g. there is no way to describe how simultaneous tasks may depend on each other.

Other popular models of interactive systems are 'compositional' models, which are generally more suited to SE issues, as modular composition (manner of division of a system into subsystems) is an important SE principle. These models include Model-View-Controller (MVC) and Presentation-Abstraction-Control (PAC) (see [23] for a comparison, or [29] or [8] for an overview of variants). In these models, systems are composed of networks or hierarchies of concurrent agents, or 'interactors'. In PAC, separation is made within each interactor between the components internal data (A), presentation to and interaction with the user (P), and dynamics and communication to other PAC-modules (C). Furthermore, it prescribes communication dependencies to be hierarchical. The structure of MVC is less strict: there may be one or more models, standing for modules within the application's internals. There may be one or more view-controller component pairs communicating with the models. The controller may in turn be a model for more view-controller pairs. The view-controller pairs have a one-to-one relation to user interface objects. The view controls the presentation to the user, and the controller manages the user's manipulations.

The idea behind these models is closely related to the object-oriented paradigm, the goal of which is to enable maximum separation of concerns and re-use of standard components. The models by themselves are not formal, but can be formalised easily by describing the components in a formal language [29]. Usually they are used only to model the dependency structure of object-oriented programs, the components having a one-to-one relation to objects. However, even the SE benefits of prescribing such fine-grained composition, and emphasising composition rather than dynamics, are still uncertain, as they are for object-oriented methods in general [11] [26] [13] [1]. One of the problems that have been reported is the difficulty of following the control flow of an object-oriented program, even if the program has no processes runing in parallel. Explicit emphasis on dynamics, not inherent to interactor models, may greatly aid SE development.

Summarising, we can say that the aspects emphasised by the HCI and SE parties are different: one emphasises control flow, while the other emphasises data flow. These views may turn out to be hard to compare because of this [17], and it may result in different aspects to remain underspecified, formal precision notwithstanding. This may be a problem, since the views may lead to incompatible specifications as well, as is indicated in [38]. In this article, it is even claimed that usability is incompatible with re-usability. At the least, it would be useful to have the system and task models in the same formal notation, so they can be compared and verified more easily [35] [29].

1.2 Our approach

The emphasis of this article lies not on the possibilities of formal verification of usability and other properties, but on the possibilities of a formal technique to act as a reference point through various activities of the development process. It should combine systems design, prototyping, and task modelling.

We have chosen to base our formal technique on process algebra, in particular, Communicating Sequential Processes (CSP) [21]. CSP by itself is compositional, and can easily be combined with interactor models, as is shown in Markopoulos's work. Markopoulos combines ADC (a variant on PAC) with LOTOS (LOTOS [6] is an international language standard, and an extension of CSP) [29] [30]. In CSP, all dynamics are also described explicitly. It is close enough to an actual program to be easily executable, lending itself to prototyping approaches. Markopoulos even suggests that hierarchical task models may be expressed naturally in process algebra, though a more detailed account of this would be desirable. However, Markopoulos ran into some of the limitations of LOTOS for user interfaces: in particular, the UI prototyping capabilities of the LOTOS tools are limited.

Our own approach is to use CSP as a basis, but not limit ourselves to standard CSP toolkits, which are generally designed for hardware and software verification. Instead, we tailor our own formalism and additional tools for use with VE development. We are developing this technique by starting off with the simplest form of CSP, and exploring its possibilities and limitations by trying to model the VMC in some detail, so as to obtain a larger than toy-size example. The VMC is considered to be a sufficiently rich ex-

ample, also containing a natural language dialogue agent and multimodality. In comparison with interactor models, the compositional granularity of our technique is slightly coarser: an interactor (as comparable to a PAC agent, an M component or a VC pair) corresponds to one CSP process. Unlike interactor models, this technique does not prescribe specific compositional dependencies, enabling experimentation with the technique and with interaction strategies.

2 CSP as a modelling language

Specification in general has a number of relevant aspects that we will discuss here.

- A specification should be *modular*. This means that a specification of a large system should be composed out of smaller specifications. The smaller parts should be relatively independent, and should be meaningful specifications in their own right. Note that the task hierarchies and the software module compositions discussed in section 1 are both modular, though they are meaningful in different ways.
- In section 1 we have identified the need for a close mapping of a specification to a concrete and executable system for the design of interactive systems. However, a specification should also allow for *property oriented* (or *declarative*) specification styles. In many cases, the specification process starts out with a rather loose set of requirements that all should be satisfied. A property oriented specification style matches this situation, because it allows one to formalize each of these requirements in turn, and then to conjoin them.

Process algebras are related to *state-based* specifications. These have been used for a very long time and go back to finite state automata. Especially when limited to finite state specifications, there is an impressive tool support for automatically verifying system properties, for instance by tools like SPIN, Verilog, etc. Not all system requirements are specified easily or naturally in terms of states, however, and so it is important that state-based specifications can be combined with, for instance, behavioral specifications.

The theoretical basis of process algebras, like CSP, can be found in the theory of *labeled transition systems* (LTS). A LTS may in turn be described in terms of a set of logical constraints. Basically, a LTS is a state-based system. It is a graph with a set of nodes Q that are called *states*, and with labelled edges which represent a *transition relation* t. We start off with a small example, taken from our VMC specification. Here, a user can walk around within the confinements of a building and interact with two other agents. A natural requirement for interaction is physical proximity of the user to the agent. We might have additional requirements, for example, the requirement that the user is looking at the agent.

This situation, although fairly simple, brings forward some typical issues already. We wish to work with a concept like "proximity requirements for communication" in an abstract way, without going into great detail how this is implemented, because:

- The precise definition of proximity might change over time, for example by incorporating a notion of viewing direction.
- The precise implementation might involve too much detail to specify practically.
- For the current version of the VMC, all that the agents react to is whether communication between the user and some agent is proximity-enabled or not.

The solution here is *not* to specify all minute detail, but rather to abstract from it: we specify only what is relevant for modelling the dynamics of most of the processes and agents in the VMC. In our example, the proximity status of the user may be modelled by the following three-state LTS (see figure 1):

1. In state `UserPos1`, the user is not in the neighbourhood of any agent,
2. In state `UserPos2`, the user is in the neighbourhood of the Karin agent,
3. In state `UserPos3`, the user is in the neighbourhood of an information board (abbreviated with "ib").

This rather minimal notion of position is all that is necessary to model the user's proximity to the agents. Note that, despite the simplicity, we have already specified something relevant: *the user cannot be both in the neighbourhood of Karin and the information board at the same time.* This property may be used for instance to prevent ambiguity when the user poses a question to an agent.

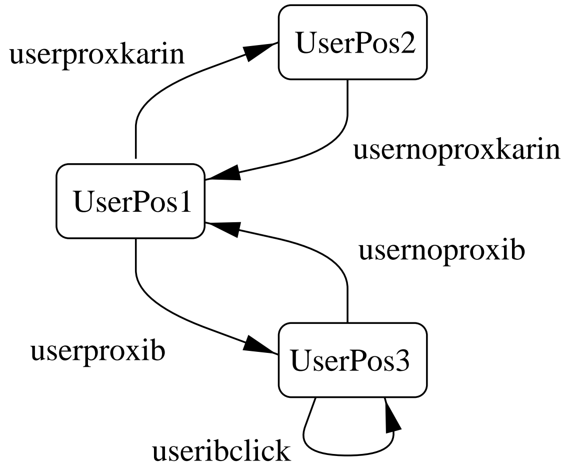

Figure 1: LTS specifying proximity relation

The state diagram also shows all possible transitions. The transitions are labeled by event names: each transition corresponds to an occurrence of the event given by its label. For instance, there is a transition from `UserPos1` to `UserPos2`, signifying an event `userproxkarin` where the user moves into the neighbourhood of the Karin agent. Note that the diagram also shows a transition labeled `useribclick`, which does not result in any state change. However, the fact that `useribclick` is only possible in `UserPos3` signifies that this event is only enabled in this state. So, a LTS can be viewed as a constraint specification, constraining which events are possible in what order.

A typical CSP system consists of a conjoinment of many such LTSes. In terms of logic, this conjoinment is a logical conjunction of a set of requirements, i.e. all these requirements should be satisfied. In terms of CSP, this is called the *parallel composition* of processes. This is in analogy with parallel execution on parallel machines, but this does not mean that the CSP process specifications have to be mapped to a parallel implementation, as long as the implementation satisfies the given constraints.

The LTS representation given here is graphical. Graphical representation is attractive, but has some disadvantages:

- For large numbers of states, a diagram showing all states does not provide much information,
- Editing and automatic processing of textual specifications is easier.

Therefore, we also use a textual representation of labeled transition systems, in the form of CSP processes. The diagram above is represented thus:

```
UserPos1 =    (userproxkarin -> UserPos2)
           [] (userproxib -> UserPos3),

UserPos2 = (usernoproxkarin -> UserPos1),

UserPos3 =    (usernoproxib -> UserPos1)
           [] (useribclick -> UserPos3)
```

In this simple specification, there is a direct correspondence between the diagram and the CSP text: each state corresponds to a so-called process definition of the form $X = Process$, where X is some process name, like `UserPos1`. $Process$ can have the form $a \rightarrow P$, where a is some transition label, and where P is itself an expression defining a process. This denotes the process that first does action a and then behaves like P. This construct is called the *action prefixing*. $Process$ may also have the form of a *choice* construct $P_1 [] P_2$. For instance, `(userproxkarin -> UserPos2) [] (userproxib -> UserPos3)`, denotes a process that (initially) has the choice between a `userproxkarin` and a `userproxib` action. As soon as one of these two actions occurs, the choice has been made and the process behaves like either `UserPos2` or `UserPos3`.

In the example above, the action prefixing construct is only used in the form $a \rightarrow X$, where X is some process name. In this case, the process definitions have a one-to-one correspondence to states in the LTS. However, any expression can follow the action. For instance, we could replace the fragment `(userproxkarin -> UserPos2)` by `(userproxkarin -> usernoproxkarin -> UserPos1)`. In this case, the LTS has more states than process definitions.

The *parallel composition* of processes P_1 and P_2 is denoted by $P_1 \parallel P_2$. For example let us consider (a sketch of) the Karin process:

```
Karin =
    (userproxkarin -> openwindows ->
                      initkarin -> Karin)
 [] (usernoproxkarin -> closewindows ->
                      exitkarin -> Karin)
```

This specification *in isolation* specifies two distinct sequences of actions, respectively following a `userproxkarin` or a `usernoproxkarin` action. Now, let us combine this with the `UserPos1` process into a system:

Figure 2: CSP diagram of Karin and user

Figure 3: LTS of Karin and user

```
Vmc where

Vmc  =  UserPos1  ||  Karin
```

Parallel specifications may also be represented in a graphical format, see figure 2. The figure was automatically generated from the specification with help of the daVinci graph layout program. The parallel composition is represented by a special `||` node, while the two subdiagrams representing `UserPos1` and `Karin` are regular LTS diagrams.

The resulting system is constrained by the constraints of both processes. Each process only places constraints on the actions that it actually uses. So, the processes only influence each other through those action labels that are shared by these two processes. Here, the shared action labels are `userproxkarin` and `usernoproxkarin`. In terms of CSP, shared action labels may also be called *communication channels*, and occurring actions may be called *signals*. Shared actions may only occur when the two processes execute it *jointly*. For instance, if one of the two processes is in a state where an `userproxkarin` action is not enabled, then the action cannot occur at that moment. For instance, after a `userproxkarin` action, the `UserPos1` process, which has turned into a `UserPos2` process, is able to do a `usernoproxkarin` action. However, since the `Karin` process does *not* enable it, the `UserPos` process must *wait* at this moment. The `Karin` process is now able to do an `openwindows` action, followed by an `initkarin` action. These two action can happen *independently*. That is, the `Karin` process will execute these actions without synchronization with the `UserPos` processes. In all likelyhood though, there will be other processes in the system as a whole, that at this point *will* participate in the `openwindows` and `initkarin` actions. After these two actions have occurred, the `Karin` process is back at the initial state where a choice can be made between the `userproxkarin` and the `usernoproxkarin` actions. Here, the `UserPos` processes enforce that the `usernoproxkarin` action will take place, since the other action is disabled by the `UserPos` processes. The possible actions of the two systems combined may be modelled by a single big LTS, see figure 3. Note that, when there are many processes, such LTSes tend to become too large to understand, though they are generally small enough for exhaustive automatic verification.

There are two basic process terms in CSP that finish the execution of a process: **skip** and **stop**. The difference lies in what happens *after* the pro-

cess has finished. Termination by means of a **stop** process denotes that all activity of the process ceases. Termination by means of **skip** denotes that a process that sequentially follows the current stage is started. This is denoted by *sequential composition* of the form $P_1 ; P2$, where P_2 is started as soon as P_1 terminates by executing **skip**. In practice, a sequence of actions like $(a_1 \rightarrow a_2 \rightarrow \cdots \rightarrow a_n \rightarrow \textbf{skip})$ is abbreviated as: $(a_1 \rightarrow a_2 \rightarrow \cdots \rightarrow a_n)$.

We introduce CSP operations for *renaming* and for *hiding* actions from a process. As an example, consider the following simple process that records whether a certain door is open or closed:

```
Door = (open -> close -> Door)
```

Compare this with a very similar process which specifies that the information board could be switched on or off:

```
OperateIB = (on -> off -> OperateIB)
```

In fact, basic patterns like these occur many times. An easy way of specifying them is to first specify the underlying *generic* pattern, thus:

```
TwoStates = (a -> b -> TwoStates)
```

Then, one uses the renaming operator of the form $P[d/c]$, denoting that action c of process P is renamed into d. In the example cases, we would define:

```
Door = TwoStates[open/a,close/b]
OperateIB = TwoStates[on/a,off/b]
```

Actually, many software components that one finds in for instance Java libararies for user interfaces can be specified by this sort of simple finite state processes. A TwoState process as above for instance corresponds directly to a (two-state) knob on a user interface.

Finally we discuss the CSP *hiding* operation of the form $P\backslash a$. The semantics of the operation is that all a actions inside process P are made invisible from outside. Hiding can be used to hide low-level details of a system that are deemed to be of less importance. In general, hiding can be used to create various *views* on a system.

For instance, actions like `userproxkarin` can be considered low-level actions from the viewpoint of the user interface, which we don't want to see. However, we *do* want to see the effect of these actions, such as the fact that Karin's windows and the information board's windows cannot be open at the same time. This may be achieved by a CSP term of the form $VMC\backslash userproxkarin$. A second example of a view is a focus on actions related to the information board only, which is achieved by hiding all actions not related to the board.

We summarize the CSP language constructs in the following table, which provides the grammar of the language:

Processes
$P ::= \textbf{stop} \;\mid\; \textbf{skip} \;\mid\; a \rightarrow P \;\mid\; X \;\mid$
$P_1 \,[]\, P_2 \;\mid\; P_1 \parallel P_2 \;\mid\; P_1 ; P_2 \;\mid\; P[b/a] \;\mid$
$P\backslash a \;\mid\; P \textbf{ where } X_1 = P_1, \ldots, X_n = P_n$

Sequential composition, parallel composition, and choice are all associative operations. This means that we may specify, for instance, $P_1 \parallel P_1 \parallel \cdots \parallel P_n$, without semantic ambiguity. Finally, we assume that parallel composition is an operator with lower priority than choice, which in turn has lower priority than sequential composition.

Most versions of CSP are more extensive than this. Extensions usually include value passing (data values may be passed from one process to another through a channel) and guarded actions (a guard is a Boolean function of data values, which, if false, prevents a transition to occur). Actually, these are just shorthand notations for plain CSP definitions. Note that data passing is generally not modelled when the data does not influence the dynamics of the system: the processes 'go through the motions' of passing data, but no data is passed.

2.1 Execution and prototyping

The current executable version provides hand execution, where the system can be stepped through by hand. By adding some directives to the definition of each CSP process, specification of a mapping between CSP signals and user interface calls or events may also be defined. This setup is similar to the scenario proposed by [2].

The coupling between CSP and the interface is as follows (see figure 4): a subset of the CSP processes correspond to user interface (UI) components. Within these processes, a subset of channels correspond to UI functions and events. The user can be seen as an observing CSP process, having as its alphabet the union set of all UI chan-

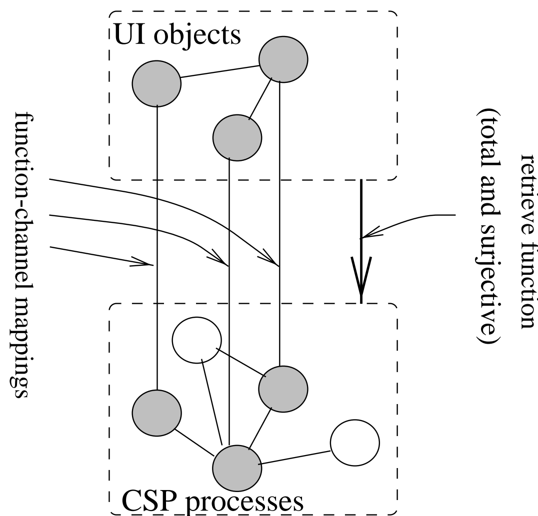

Figure 4: Architecture

nels. A number of standard UI component types are available, each having a set of standard functions and user-generated events.

Direct dependencies between interface components may also exist. For example, in the CSP descriptions that follow, windows are assumed to be independent, while in reality, they may obscure each other. If a relatively complete formal coverage is desired, we should show that these dependencies do not lead to undesirable situations. This can be done by describing a retrieve function $\rho : UI \rightarrow CSP$ [39], which defines which concrete (UI) state maps to which abstract (CSP) state. This function should be total (i.e. all UI states have a defined CSP state), and preferably surjective (each state that is possible in CSP should also be possible in the UI). The surjectiveness is necessary to guarantee that reachability properties remain valid: otherwise, some states exist that are reachable within the CSP specification, but could never be reached through the UI.

In our example, we could assert totalness and surjectiveness by proving that any window may always be moved in such a way that all of its contents are visible and that moving a window can be done without generating unwanted signals, or that windows are in fixed positions while not obscuring each other. Note that this kind of specification is at a rather high level of detail. The examples of interactor models usually found simply assume that windows are reachable.

The mapping between channels and UI functionality may be defined by means of special directives, which are added to the declaration of the processes that correspond to UI components. The current notation is somewhat ad-hoc, but it illustrates the general principle well. A list of directives, placed between curly braces, may be declared before the body of each process declaration. Among the directives available are the following:

`type`	`[<func>]`	component type
`input`	`{<chan>}`	channels from system to user
`output`	`{<chan>}`	channels from user to system

Within the `<chan>` directives, a list of channel names may be defined, each of which is followed by a body in which more directives may be placed. These include the following:

`receive`	`[<func>]`	function to call when signal occurs
`init`	`[<func>]`	function that sets up an interface listener which generates the signal

Within the `<func>` directives, a UI function with optional parameters may be supplied.

The prototyping system is implemented by means of two communicating Unix processes: a C program implementing the CSP engine, and a Tcl process managing the user interface. The engine communicates the set of possible transitions that could occur given the current state (this set is called the nextset), then the Tcl process communicates its choice back to the engine, etc. The execution model is as follows: from all channels in the nextset that do not correspond to user output channels, one is randomly chosen. This amounts to 'flattening' parallel execution into random sequential execution, which is an established model [4]. If the channel corresponds to UI feedback, the UI is updated accordingly. If only user output channels remain, the user may generate signals by manipulating the interface.

3 Example models

In this section, we discuss some of the potential possibilities and problems of CSP as a modelling language by means of examples. The examples are centred around the VMC, which is a VE modelled after the music centre building in Enschede. The VMC project [34] is meant as a testbed for research in experimental interaction, such as natural language, virtual reality, and multimodal interaction. The project aims to be Web-based, i.e. to be accessible through standard Web-browsing

tools. One of the agents in the VMC is a natural language dialogue system (originally called Schisma), which can be queried for information about performances, and which can handle reservations.

The examples given here include parts of the global system model, a technical design detail, and a task model.

3.1 System model

In the model of the VMC specified here, we have tried to include some of the most interesting agents found in the real VMC. This includes an information board, which displays information about today's performances. An attempt is also made to specify the Schisma dialogue agent, including its dialogue manager. The specification leaves out the lowest level of detail: not specified are mouse movement, the details of text input and parsing, the navigation interface, and the details of appearance of VE objects, window dressing, fonts, etc. The specification follows the actual implementation reasonably faithfully, though modelling implementation details is not considered an objective of the specification. The system consists of a number of agents, each of which has a presence in the world. The Schisma system (which is called Karin in the VE) is subdivided further into a dialogue engine, some auxiliary user interface objects, and some communication buffers. See figure 5 for a system overview.

The processes at the top level are agents within the VE. The `User` process also stands for the user's view and influence within the environment: the user can move around, generating proximity notification signals, to which the agents may react. In case more users are added, they would appear in the specification as `User` objects. Within the `User` process, the precise constraints on the user's influence are defined.

```
User {...} = UserPosN,

UserPosN =
     (userproxkarin     -> UserPosK)
  [] (userproxib        -> UserPosI)
  [] (userproxdoorfront -> UserPosDF)
  [] (userproxdoorinner -> UserPosDI),

UserPosK = (usernoproxkarin -> UserPosN),

UserPosI = (usernoproxib    -> UserPosN),

UserPosDF =
```

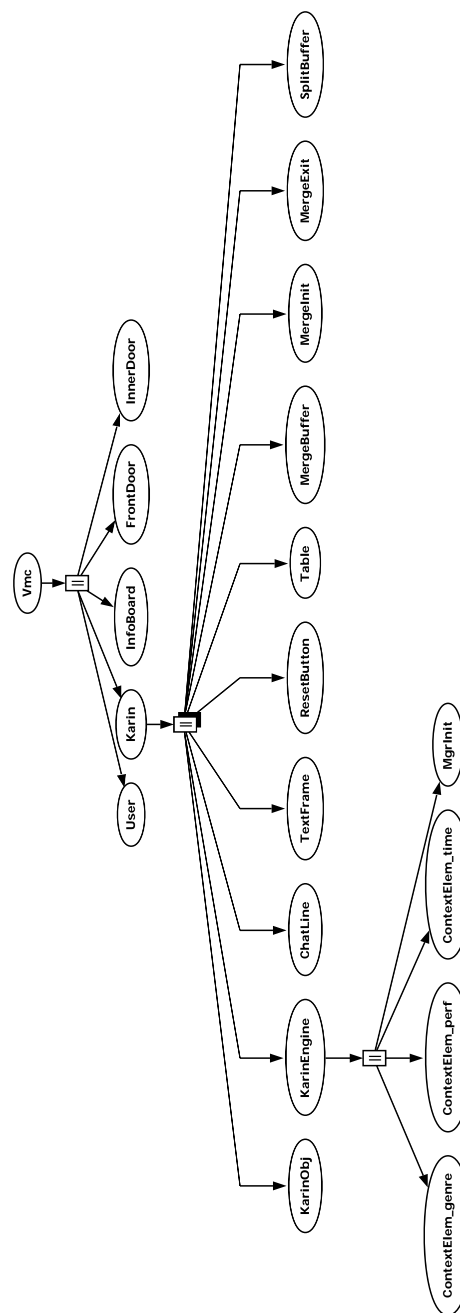

Figure 5: Overview of processes in the VMC Graphical representation of the VMC process declarations. The leaves of the tree contain the state automata (not shown). Note that the communication dependencies are not constrained by the tree structure, but may be arbitrary.

```
  (usernoproxdoorfront -> UserPosN),

UserPosDI =
  (usernoproxdoorinner -> UserPosN)
```

From the system's point of view, no more than a few discrete positions are necessary to model the user's location: agents may be notified of the user's entering or leaving their proximity, modelled as `userprox...` and `usernoprox...` signals. The user starts in front of the building, which corresponds to `UserPosN`. In this specific case, the agents are all at a sufficiently large distance from each other, so that the user can never be near two objects at the same time. `User` has only five positions, one for each agent s/he can be near, and one in which no agent is near.

Constraints such as these have to be specified if we require surjectiveness of the retrieve function: by specifying precisely where all agents may go, we can be sure the specification corresponds precisely to all possible situations. The question remains how such constraints could be specified in more general cases, for example in VEs in which the agents are able to move or a more involved notification scheme is used.

The Karin agent consists of Karin's manifestation in the world (`KarinObj`), Karin's dialogue manager (`KarinEngine`), a number of buffers for communicating with other processes, and a couple of windows used for communicating with the user. The user can also reinitialise Karin using a reset button, and Karin can display query results in a table. The four buffers are needed for multi-way communication: there are several processes that communicate with `KarinEngine`, but do not communicate with each other. If these would all try to talk with `KarinEngine` over the same channels, *all* processes would be forced to synchronise their communication. The buffer implementation allows each process to write data independently, and even overwrite the data of another before `KarinEngine` could have read it: some data may be lost due to concurrent access. Note that the same problem was found in the real VMC while specifying, and is made explicit by the specification. If this data loss is not considered acceptable, the buffers could be made to block multiple consecutive writes, and deadlock conditions resulting from blocked buffers could be detected and eliminated.

The process `KarinObj` defines the presence of Karin and Karin's reaction to the proximity of the user.

```
KarinObj {
  type [object(350,150,"karin.gif")]
  output{
    userproxkarin   {init[setprox()]   }
    usernoproxkarin {init[setnoprox()] }
  }
}
 =    (userproxkarin -> openwindows
               -> initkarin1 -> KarinObj)
   [] (usernoproxkarin -> closewindows
               -> exitkarin1 -> KarinObj)
```

The UI directives define that `KarinObj` is a VE object, and set up interface listeners that generate the proximity signals `userproxkarin` and `usernoproxkarin` when the user approaches or leaves. The process reacts to these signals by respectively setting up or closing down the dialogue manager and the windows. Note the '1' in `initkarin1` and `exitkarin1`: this means that the process wants to signal `initkarin` and `exitkarin` to the dialogue manager, but the signal is buffered (in this case, through `MergeInit` and `MergeExit`). Other processes should signal their init and exit signals through other channels, named `...karin2` etc.

The processes `ChatLine`, `TextFrame` and `Table` are windows, which have two main states, `...Open` and `...Closed`. Only in the `...Open` state may the window display information and accept user input. As an example the `Table`-process is given:

```
Table {
  type [window]
  input{
    usertableopen  { receive [open()]  }
    usertableclose { receive [close()] }
    usertablerefresh {
      receive [showtext("Table filled")]
    }
  }
  output{ userclick {
      init [createbutton("click")]
  } }
} = TableClosed,

TableClosed =
     (opentable -> usertableopen ->
                              TableOpen)
  [] (closetable -> TableClosed),

TableOpen =
     (opentable -> usertablerefresh ->
                              TableOpen)
```

```
  [] (closetable -> usertableclose ->
                              TableClosed)
  [] (userclick -> textout2_perftime ->
                                TableOpen)
```

The table lists performances, which can be clicked on to select a performance. The information that should be displayed in the table is always supplied through the `opentable` channel. A user click results in a buffered signal `textout2_perftime` to the dialogue engine, communicating the information that specifies that performance.

The bulk of Karin's behaviour is specified in `KarinEngine`. The engine consists of the dialogue manager, starting at `MgrInit`, and some variables for keeping track of dialogue context. The dialogue manager is a simplified model, though it contains the main principles: keeping track of dialogue context by means of data obtained from previous utterances, and the manager itself is essentially state-based.

```
KarinEngine =    MgrInit
              || ContextElem_genre
              || ContextElem_perf
              || ContextElem_time,

MgrInit =  (initkarin -> textin_hello ->
                                MgrClear),

MgrClear = (delperf -> deltime ->
                 delgenre ->  MgrWait),

MgrExit =
   (exitkarin -> closetable -> MgrInit),

MgrWait =
   (textout_genre -> addgenre ->
              delperf -> MgrAnswTable)
 [](textout_perf -> addperf -> MgrPerf)
 [](textout_time -> addtime -> MgrTime)
 [](textout_perftime -> addperf ->
            addtime -> MgrAnswInfo)
 [](textout_book -> closetable ->
                                 MgrBook)
 [](textout_yes -> textin_error ->
                                 MgrWait)
 [](textout_no -> textin_error ->
                                 MgrWait)
 [] MgrExit,

MgrPerf =   (gottime -> MgrAnswInfo)
         [] (nogottime -> MgrAnswTable),

MgrTime =   (gotperf -> MgrAnswInfo)
         [] (nogotperf -> MgrAnswTable),

MgrAnswInfo =
           (textin_tellinfo -> MgrWait),

MgrAnswTable = (textin_telltable ->
                     opentable -> MgrWait),
```

After some initiating actions, the dialogue manager arrives at `MgrWait`. This is the state in which Karin accepts input from the user, and reacts to it by providing information. There are two kinds of information: detailed information about one specific performance, or a list of performances. In this simplified model, it is assumed that a performance is fully specified by supplying both a performance name (`textout_perf`) and a time (`textout_time`). A list of performances can also be obtained by supplying a genre (`textout_genre`). The rest of the states are for determining what to answer. In case a query results in a list, the table is opened to display it. When the user wishes to book for a performance (`textout_book`), the manager goes into a separate booking mode `MgrBook` (not shown), which is analogous to `MgrWait`. The manager returns back to `MgrWait` when a performance has been specified and confirmed, or the booking is cancelled, or the user makes an irrelevant utterance.

Specification of the dialogue manager in simple CSP is somewhat verbose, but it is feasible to describe the complete dialogue manager this way, preferably aided by some shorthand notations for managing the context variables. This dialogue manager is finite-state (a finite-state automaton, having a set of pre-programmed states and transitions) rather than plan-based (taking actions on the basis of a hierarchy of goals and plans, created by a 'planning' engine) [24]. However, it may also be possible to describe plan-based dialogue managers in CSP. This could be done by modelling plans as processes, and defining each plan as a parallel composition of sub-plans, which is analogous to the task model found in section 3.3. Plans may also be defined recursively, just like tasks in a CSP task model [29].

In our opinion, this kind of specification has been useful for describing the system up to an interesting level of detail, and for uncovering concurrency issues: apart from the buffering problem, some other small bugs in the real VMC were found while specifying. For example, the user could select from an empty table, resulting in a nonsense reaction, and the reset button has a

strange effect when clicked while Karin is still processing.

A VMC “prototype” can also be generated easily, with help of the (still limited) UI component library. See figure 6 and figure 7 for a comparison of the prototype with the real system. The navigation screen shows an overhead 2D map, rather than first-person 3D. The user, indicated by the label “userobj”, can be moved around. If the user comes close to an agent, a separate window is opened, which shows a detail view of the agent. Instead of querying Karin by means of text input, the user can select his/her query by selecting from a column of buttons, each standing for one type of information that can be entered as text in the real VMC. Even though it is still very simple, the prototype could for example be used to examine visibility and layout issues, and get an idea of the system’s “feel”.

3.2 Detailed description of multimodal output

This section illustrates how more detailed and technical design issues could be discussed using the language. In the new version of the VMC, Karin’s output is not limited to text. Speech and mouth movement are added, which have to be synchronised. To do this, the text is first converted to phonemes and speech. The speech is played as an audio sample, and each successive phoneme is converted to mouth animation while the sample is playing. The system is shown in figure 8.

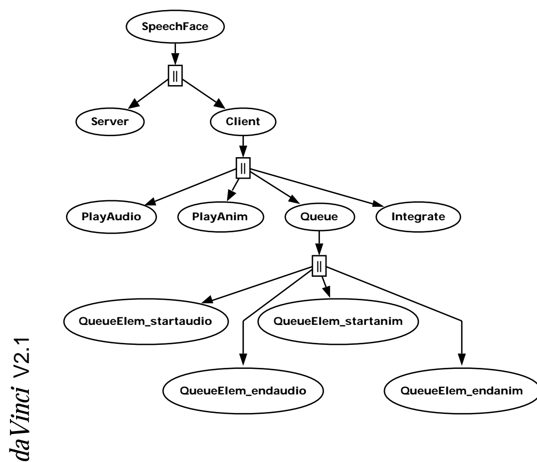

Figure 8: Speech-face system

The system is internet-browser-based, which means there are some serious technical constraints, as will become clear below. The text-to-speech program is running on a separate machine:

```
Server =    (sentence -> texttospeech
             -> Server)
         [] (loadaudio -> Server)
         [] (loadphonemes -> Server)
```

First, a sentence is sent to the server (`sentence`), which is then converted to speech and phonemes (`texttospeech`), which can then be downloaded by the client separately (`loadaudio` and `loadphonemes`).

The standard animation player component can be described as follows:

```
PlayAnim = (playanim
  -> startanim -> queuestartanim
  -> endanim -> queueendanim
  -> PlayAnim)
```

When giving the command to start playing (`playanim`), the animation starts (`startanim`, an internal event). This is then notified to a separate `Queue` process through a signal `queuestartanim`. `Queue` then enables a signal `notifystartanim` to be read once by other processes. The end of the animation (`endanim`) is similarly notified through `queueendanim`.

The audio player component might have had exactly the same structure as `PlayAnim`. However, a bug was discovered which causes the player to behave badly when a new sample is loaded while the old one is still playing. In other words, the audio player somehow participates in loading samples, and any attempt to load new samples while it is still playing results in undesirable behaviour (in the following specification, deadlock).

```
PlayAudio = (loadaudio
  -> (    PlayAudio
       [] (playaudio
            -> startaudio
            -> queuestartaudio
            -> endaudio
            -> queueendaudio
            -> PlayAudio)
     ) )
```

The technique currently used to synchronise these processes is as follows:

```
Integrate = (sentence -> loadphonemes
```

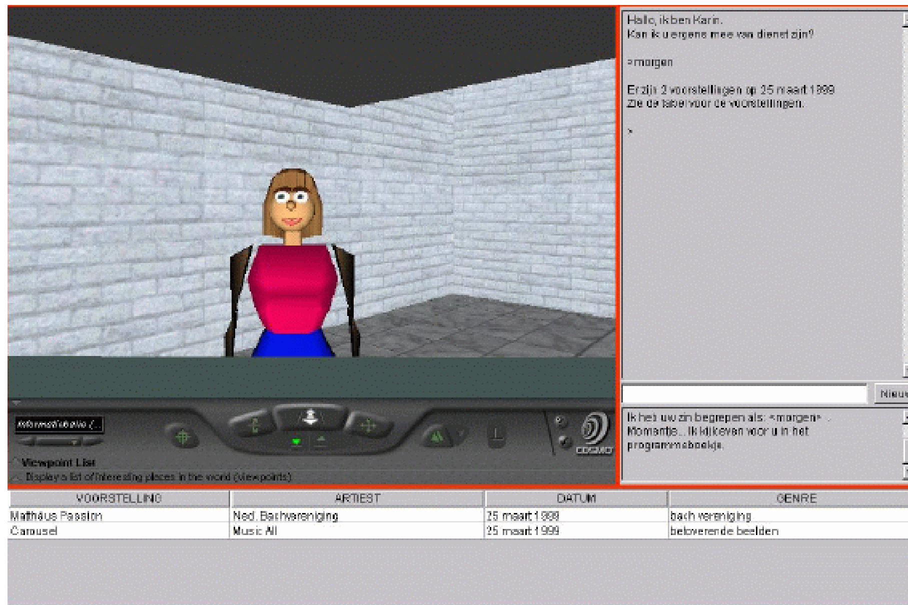

Figure 6: The real VMC

This screengrab shows the user talking to Karin. The user has just entered a date in the text entry field (the white bar on the right), and Karin replies (top right) with help of a table (bottom).

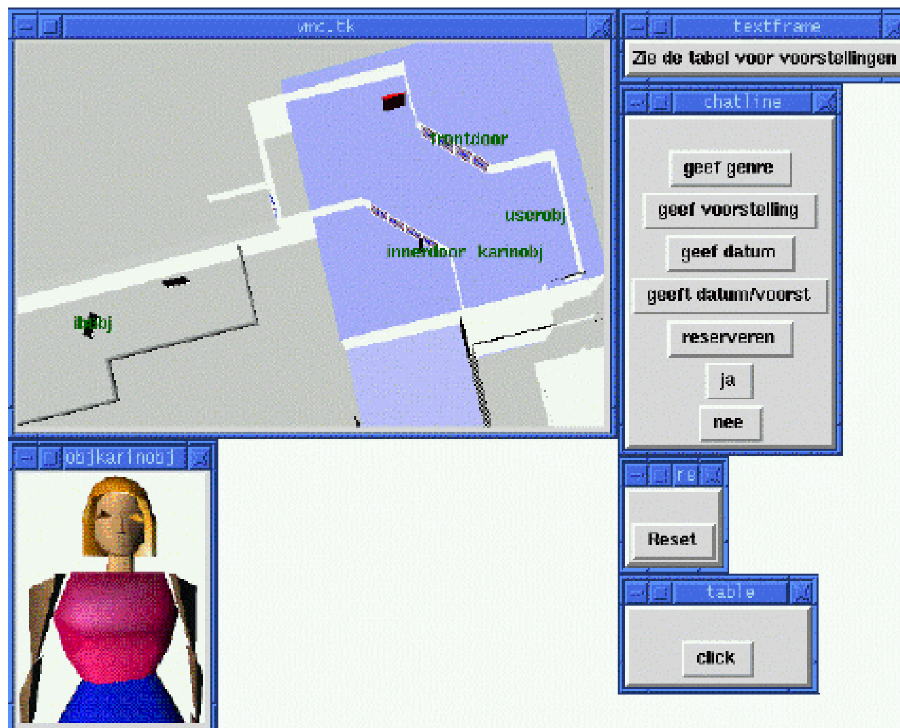

Figure 7: Generated VMC prototype

This screengrab shows the same situation as figure 6. At the top left is the navigation window, top right is Karin's reply, middle right is the text entry window, bottom left is Karin's detail view.

```
  -> loadaudio -> playaudio
                          -> IntVisWait),

IntVisWait = (notifystartaudio
                          -> IntVisPlay),

IntVisPlay =
     (playanim -> notifyendanim
                          -> IntVisPlay)
  [] (endvisemelist -> IntVisTerm),

IntVisTerm = (notifyendaudio
                           -> Integrate)
```

However, the time between `playanim` and `notifyendanim` turned out to be longer than the specified viseme duration, resulting in a cumulative buildup of delay. The first proposed solution was to split the samples into smaller parts. However, the delay between `sentence` and `playaudio` (the downloading turns out to be slow) causes unacceptable delay between the samples. It is not possible to load the next sample while the current one is playing either, because of the bug in `PlayAudio`. A limited solution would be to react on `notifyendaudio` immediately to stop the mouth early. The solution currently implemented is to measure time immediately after `notifystartaudio`, and to calibrate the duration of successive visemes according to the time that really elapsed as opposed to the viseme duration specified.

3.3 Task analysis

CSP can be used for specifying task hierarchies similar to basic forms of HTA reasonably naturally: each task and subtask may be specified as a process, and each basic operation may be specified as a channel. Loops and interleaving tasks could also be described easily. The task model can be tested with the system by parallel composition with the system. CSP also has some interesting possibilities for modelling more detailed cognitive aspects, such as task performance and memory load, as in task modelling languages like GOMS [16] and ETAG [14]. We list some of the possibilities here.

- In GOMS and ETAG, short-term memory is modelled as a goal stack, containing a list of all goals and subgoals currently being considered by the user. These may also contain parameters needed to perform the task. In CSP, this may be modelled by the processes currently running. In extensions of CSP with value passing, the parameters may be modelled by the values bound by each process.
- In GOMS and ETAG, choice between alternative possible strategies is done by means of selection rules. These rules specify whether which action should be taken by means of a Boolean function of the user's memory, or, possibly, of the system's feedback. Choice in basic CSP is simply by the choice operator, which could model both types of choice. CSP extensions with value passing generally allow choice to be made using an arbitrary Boolean expression as well.
- ETAG also specifies a domain model, describing the objects in the task domain, and their relations. In a CSP model, this roughly corresponds to the relation between channels and processes, and between different processes in the system model. However, no arbitrary inter-object relations may be described.
- Extensions of GOMS, such as EPIC [25], also enable highly detailed cognitive performance prediction, using a model of concurrent cognitive subsystems or critical path analysis for tasks that can be done simultaneously. This is especially interesting for modelling the effect of multimodality on user tasks. In CSP, tasks that may be interleaved can be described quite naturally by means of parallel composition. Cognitive subsystems may also be modelled by describing them as separate CSP processes. Real-time extensions of CSP [9] also enable specification of durations, which could be used for critical path analysis.

Here, we give an example of a task model. Note that the task is only one example of the tasks that a user may be trying to achieve with the VMC system. In this task, the user wishes to go to the theatre today, and is trying to select a suitable play. It is assumed that the information needed to select the play can only be obtained from the information board or by asking Karin for details about the performance. The task hierarchy is as follows:

```
Goal = Query ; Book,

Query = QueryIB [] QueryKarin,

QueryIB = GotoIB ; GetInfoIB,
```

```
QueryKarin =
 GotoKarin ; GetPerfList ; QueryEachPerf,

QueryEachPerf =
  SpecifyPerf ; QueryEPRead ;
  (    (seemore ->    QueryEachPerf)
    [] (seenenough -> skip) ),

Book = (    (GotoKarin ; PerformBook)
         [] (PerformBook) )
       ; Confirm

PerformBook  =
     (SpecifyBook ; SpecifyPerf)
  [] (SpecifyPerf ; SpecifyBook),
```

The goal consists of two subgoals: obtain the information (`Query`) and book for the play (`Book`). At this level of specification, there are several choices available to the user: in `Query`, s/he can query the information board (`QueryIB`) or Karin (`QueryKarin`); in `PerformBook`, s/he can first specify that s/he wants to book, and then specify which play to book, or do it the other way round. When querying details in `QueryEachPerf`, the user repeats the subtasks until s/he has seen enough, which is determined by the channels `seemore` and `seenenough`, which are internal to the user. When trying to book, it is possible that the user is still at the information board and has to walk to Karin first. The feedback from the system can be used to determine which one is applicable. Note that two tasks occur at different places, namely `SpecifyPerf` and `GotoKarin`. The following bottom-level tasks exist for the information board:

```
GotoIB =    GotoIB2
         [] (usernoproxkarin -> GotoIB2),
GotoIB2 = (userproxib -> skip),

GetInfoIB = (useribclick
             -> useribshowinfo -> skip),
```

Note that the navigation task `GotoIB` is versatile: the user will arrive at the information board regardless of where s/he is, by reacting to system feedback. The query information is obtained by receiving the signal `useribshowinfo`. If the user were not interested in this information, s/he could simply choose to ignore the signal. The bottom-level tasks for Karin are:

```
GotoKarin  =
     GotoKarin2
  [] (usernoproxib -> GotoKarin2),
GotoKarin2 = (userproxkarin -> skip),

GetPerfList = (usertyped_time
              -> usertableopen -> skip),

SpecifyPerf =
     (userclick  -> skip)
  [] (usertyped_perftime -> skip),

QueryEPRead =
  (usershowtext_tellinfo -> skip),


SpecifyBook  = (usertyped_book -> skip),

Confirm = (usershowtext_confirm
          -> usertyped_yes
          -> usershowtext_done -> skip),
```

`GotoKarin` is analogous to `GotoIB`; the other tasks are mostly obvious. Note that, when querying Karin for specific performances (`QueryEachPerf`), the user can obtain specific details either by clicking on the performances table (`userclick`) or by keying in a specification of the play (`usertyped_perftime`). In case the table is not open (this happens when the user has chosen to query the information board and is now trying to book), the system feedback will simply not allow the former option.

A weakness of the current model is that the user can only observe system feedback by seeing that certain output channels are enabled, or by catching input signals as they are generated. In some cases, for example when a task model requires re-reading an answer given by Karin, channels for re-reading the display should be added to the system model. For example, for each window process, different states should be added that distinguish the different texts being displayed at any particular moment, and an extra channel should be added for each state that enables the user to read that particular text. While designing the system model, such channels were never considered, because they are not meaningful from a systems designer's point of view: the system is not concerned with what happens with information after the command is given to display it. It is not even possible to detect when or if the user reads any message displayed. These extra signals could be modelled by means of variables (see below).

3.4 FUTURE RESEARCH

By means of examples, we have indicated that CSP is interesting as a central specification language for various aspects of the development process. From here onwards, several strands of research may be identified:

1. Investigating the possibilities of specifying hierarchical task models and plan-based dialogue models using process algebra. Apparently there is little research on this subject, though some examples of task models are emerging in the literature [29] [31]. However, a serious large-scale or systematic coverage seems to be lacking.

2. Extending the programming environment to include a more complete coverage of user interface functionality. The possibility of invoking arbitrary software modules from CSP, such as parsers and speech generators, is also attractive. Taken to its fullest extent, this means that CSP could be used as a central 'glue' language, glueing together separate modules, while keeping the architectural dependencies explicit, making software maintenance and experimentation easier.

   This could be done by an annotation scheme similar to the 'UI directive' scheme explained here. Apart from a less ad-hoc notation, this scheme could use a redesign: restrictions to the communication scope of the channels defined in the directives should be defined in a meaningful way, and a value passing scheme should be added.

3. Developing or adopting language extensions, shorthand notations, and verification tools that are particularly useful for VE development. The most basic ones are variables and value passing, enabling a shorter notation for some parts of the specification. System feedback through persistent modalities, as explained in the example of re-reading window contents in section 3.3, could also be modelled by means of variables. For example, a variable could stand for the state of a window, and is directly accessible by the user.

   Further extensions are interesting for VE development in particular. The ability to create multiple instances of one process is interesting for modelling multi-user systems. Real-time extensions are interesting for analysing cognitive performance in multimodal systems.

4. An interesting possibility, not yet mentioned, is automatic user data analysis in terms of the CSP signals. The sequence and duration of the signals that occur during interaction could easily be logged and analysed. This could also be useful for obtaining data for cognitive performance prediction, as is done in GOMS models.

References

[1] Ritu Agarwal, Atish P. Sinha, and Mohan Tanniru. The role of prior experience and task characteristics in object-oriented modeling: an empirical study. *International journal of human-computer studies*, 45:639–667, 1996.

[2] Heather Alexander. *Formal methods in human-computer interaction*, chapter 9: Structuring dialogues using CSP. Cambridge university press, 1990.

[3] M. E. Atwood, B. Burns, A. Girgensohn, A. Lee, T. Turner, and B. Zimmermann. Prototyping considered dangerous. In *Proceedings of INTERACT'95: Fifth IFIP Conference on Human-Computer Interaction*, pages 179–184, 1995.

[4] J. C. M. Baeten and W. P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1990.

[5] Niels Ole Bernsen. towards a tool for predicting speech functionality. *Free Speech Journal*, 1, 1996.

[6] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[7] Jonathan Bowen and Mike Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(4):34–41, 1995.

[8] Gaëlle Calvary, Joëlle Coutaz, and Laurence Nigay. From single-user architectural design to PAC*: a generic software architecture model for CSCW. In *CHI '97*, 1997.

[9] Zhou Chaochen, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information processing letter*, 40(5), 1991.

[10] Joëlle Coutaz, Laurence Nigay, Daniel Salber, Ann Blandford, Jon May, and Richard M. Young. Four easy pieces for assessing the usability of multimodal interaction: the CARE properties. In *Proceedings of INTERACT'95: Fifth IFIP Conference on Human-Computer Interaction*, pages 115–120, 1995.

[11] J. Daly, A. Brooks, J. Miller, M. Roper, and M. Wood. The effect of inheritance on the maintainability of object-oriented software: an empirical study. In *Proceedings of the 1995 international conference on software maintenance*, 1995.

[12] Simon P. Davies. Expertise and display-based strategies in computer programming. In *People and computers VIII: proceedings of the HCI '93 conference*, 1993.

[13] Simon P. Davies, David J. Gilmore, and Thomas R. G. Green. Factors influencing the classification of object-oriented code: Supporting program reuse and comprehension. In *Symbiosis of human and artifact, advances in human factors/ergonomics 20A*, 1995.

[14] G. de Haan and N. Muradin. A case study on applying Extended Task-Action Grammar (ETAG) to the design of a human-computer interface. *Zeitschrift für Psychologie*, 200(2):135–156, 1992.

[15] Norman Fenton, Shari Lawrence Pfleeger, and Rober L. Glass. Science and substance: a challenge to software engineers. *IEEE Software*, 11(4):86–95, July 1994.

[16] Wayne D. Gray, Bonnie E. John, Rory Stuart, Deborah Lawrence, and Michael E. Atwood. GOMS meets the phone company: Analytic modeling applied to real-world problems. In *Proceedings of IFIP INTERACT'90: Human-Computer Interaction*, Foundations: Cognitive Ergonomics, pages 29–34, 1990.

[17] T.R.G. Green. Programming languages as information structures. In *Psychology of programming (computers and people series)*, pages 118–137, 1990.

[18] C. A. Gurr. Supporting formal reasoning for safety-critical systems. *High Integrity Systems*, 1(4):385–396, 1994.

[19] G. de Haan, G. C. van der Veer, and J. C. van Vliet. Formal modelling techniques in human-computer interaction. *Acta Psychologica*, 78(1-3):27–67, 1991.

[20] M. Harrison and H. Thimbleby, editors. *Formal methods in human-computer interaction*, chapter Preface. Cambridge university press, 1990.

[21] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice hall, New York, 1985.

[22] K. S. Hone and C. Baber. Using a simulation method to predict the transaction time effects of applying alternative levels of constraint to user utterances within speech interactive dialogues. In *ESCA workshop on spoken dialog systems: theories and applications*, pages 209–212, 1995.

[23] Andrew Hussey and David Carrington. Using Object-Z to compare the MVC and PAC architectures. In C. R. Roast and J. I. Siddiqi, editors, *Proceedings of the BCS-FACS Workshop on Formal Aspects of the Human Computer Interface*, 1996.

[24] Arne Jönsson. *Dialogue management for natural language interfaces*. PhD thesis, Linköping University, department of computer and information science, 1993.

[25] David E. Kieras, Scott D. Wood, and David E. Meyer. Predictive engineering models based on the EPIC architecture for a multimodal high-performance human-computer interaction task. *ACM Transactions on Computer-Human Interaction*, 4(3):230–275, 1997.

[26] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen. Change impact identification in object oriented software maintenance. In *Proceedings of the 1994 international conference on software maintenance*, 1994.

[27] Ian Lewin. Formal design, verification and simulation of multi-modal dialogues. In *TWLT13: formal semantics and pragmatics of dialogue*, 1998.

[28] Kee Yong Lim and John Long. *The MUSE method for usability engineering*. Cambridge University Press, 1994.

[29] Panos Markopoulos. *A compositional model for the formal specification of user interface*

software. PhD thesis, University of London, 1997.

[30] Panos Markopoulos, Peter Johnson, and Jon Rowson. Formal architectural abstractions for interactive software. *International journal of human-computer studies*, 49:675–715, 1998.

[31] M. Mezzanotte and F. Paternó. Verification of properties of human-computer dialogues with an infinite number of states. In C. R. Roast and J. I. Siddiqi, editors, *Proceedings of the BCS-FACS Workshop on Formal Aspects of the Human Computer Interface*, 1996.

[32] Katashi Nagao and Akikazu Takeuchi. Speech dialogue with facial displays: Multimodal human-computer conversation. In *Proceedings of ACL-94*, 1994.

[33] Jakob Nielsen. *Usability engineering*. Academic Press, 1993.

[34] A. Nijholt, Arjan van Hessen, and J. Hulstijn. Speech and language interaction in a (virtual) cultural theatre. In *Natural language processing and industrial applications (NLP+IA 98) - Special accent on language learning*, pages 176–182, 1998.

[35] P. Palanque and R. Bastide. A design lifecycle for the formal design of interactive systems. In C. R. Roast and J. I. Siddiqi, editors, *Proceedings of the BCS-FACS Workshop on Formal Aspects of the Human Computer Interface*, 1996.

[36] P. Palanque and F. Paternò, editors. *Formal methods in Human-computer interaction*. Springer Verlag, 1998.

[37] G. C. van der Veer, B. F. Lenting, and B. A. J. Bergevoet. GTA: Groupware task analysis - modeling complexity. *Acta Psychologica*, 91:297–322, 1996.

[38] Hans Wegener. The myth of the separable dialogue: software engineering vs. user models. In *Proceedings of INTERACT'95: Fifth IFIP Conference on Human-Computer Interaction*, pages 169–172, 1995.

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