Synchronisation and Delay in a Formal Model of User Cognition[1]

D. Duke, G. Faconti and M. Massink

Department of Computer Science, University of York, Heslington, York, YO1 5DD, U.K.
email: duke@minster.york.ac.uk

C.N.R. - Ist. CNUCE, Via S. Maria 36, I56100 Pisa, Italy
email: M.Massink@guest.cnuce.cnr.it faconti@cnuce.cnr.it

Abstract

This work is part of a syndetic approach to the evaluation of the usability of interaction devices that takes into account the cognitive resources needed to use a device to perform particular tasks. In the syndetic approach both a cognitive model and a model of system behaviour are specified and brought together within a single framework in order to investigate their relations. The ICS model is such a cognitive model of human information processing. In this model the human information processing is depicted as a number of independent cognitive processes that cooperate by means of exchanging mental representations of the observed environment. The style in which the model is described is close to a data flow style, which is also one of the formal approaches used within Computer Science for the specification of systems behaviour. In this paper we present a data flow oriented representation of a simplified version of the ICS model in which we study the synchronisation and delay of the streams of representations flowing through the model. The data flow approach is shown to give particularly interesting possibilities to investigate the consequences of a relative difference in speed between the information processing of the human and the change of the environment in which (s)he is working.

1 Introduction

This work is part of a search for a human centered approach to the design of interactive systems. It aims at the development of a methodology that allows for a multi-disciplinary approach to system development in the early phases of the software life cycle. In those phases the most important issues are the requirement capturing and abstract specification of the system to be build. Often no prototype of the human computer interface is available at that stage which means that no experimental information can be obtained about the usability of the interface. In such a situation a theoretical model of the cognitive aspects of human information processing can be helpful to reason in an early stage about the requirements of the human

---

computer interface in order to evaluate its usability by taking into account the cognitive resources needed in the interaction with a computer.

Earlier work in this direction has been using state based notations and was aiming at the exploration of this field at a high level of abstraction [1, 17]. In other approaches theoretical models originating from psychology have only been used in an *indirect* way, see for example [8, 9, 13]. In this article we take an approach that uses a particular cognitive model, Interacting Cognitive Subsystems (ICS), in a *direct* way by formally modelling aspects of this theory in such a way that it can be combined with a system specification following a syndetic modelling approach [11]. Other work on the use of the ICS model for the evaluation and design of Human Computer Interfaces can be found in [3, 2, 11, 10]. These works do not aim at finding a complete model of human behaviour concerning the use of computer interfaces because of the complexity and variation within human behaviour. However, they shown that a formalization, i.e. a more precise mathematical model, of a cognitive theory can help to improve the theory and to raise new and different questions about human behaviour. This way the insight in and understanding of the theory is improved and this brings to ideas on how certain aspects of the theory can be used to improve the design process for the development of interfaces. The formulation of the cognitive theory in a formal framework has moreover the advantage that it can be easier introduced and explained to the computer science community.

The style of specification we use in this article is a *functional data flow* approach [15, 5, 6, 18]. This approach has been chosen because the way the cognitive ICS theory is structured resembles closely a data flow oriented approach. In this way the formalized ICS model and the more informal psychological ICS model keep being rather similar which is an advantage in multi-disciplinary discussions.

The ICS concepts we are concerned with in this article are the formal modelling of the continuous flow of information between the different subsystems and the transformations of these flows by the subsystems. Since the flow of information is not always a simple linear flow from the sensory subsystems (input) to the effector subsystems (output) there are some interesting problems that can be observed related to delay and synchronisation of the flow of information. The speed of information processing depends on the particular cognitive configuration in place, and the ability of the processes to operate on the available information. That is, the delay observed between providing some stimulus and observing a response depends for example on whether a particular reaction has been learnt (proceduralised) by the processes involved, or whether a level of reciprocal interchange of information or buffered processing of information is needed to make sense of the available information. These differences are well-illustrated by the data flow approach we use and within this formalism we can formulate solutions for synchronisation problems that are related to the delays. These solutions can be matched with results obtained from psychological experiments on human information handling.

In Section 2 we give a short description of the ICS model, followed by a data flow model specification in Section 3 that captures some of the concepts of the ICS model related to delay and synchronisation. Section 4 discusses the consequences of the data flow model and shows what is the effect of a delay caused by reciprocal loops in a cognitive configuration. The introduction of a relatively different processing speed between the model of the cognitive part and the model of the computer system part shows how a more realistic interaction can be modelled that may mirror, to a certain extend, the working of the underlying neural architecture of human cognition. Section 5 describes the interplay between delay caused by reciprocal loops and buffered transformations. In Section 6 the results are discussed and some topics for further research are outlined.

2   The ICS model — A short introduction

Interacting Cognitive Subsystems (ICS) [2, 3] is a comprehensive model of human information processing that describes cognition in terms of a collection of sub-systems that operate on specific mental codes. Although specialised to deal with specific codes, all subsystems have a common architecture, shown in Fig. 1. Incoming data streams arrive at an input array, from which they are copied into an image record representing an unbounded episodic store of all data received by that subsystem. In parallel with the basic copy process, each subsystem also contains transformation processes that convert incoming data into certain other mental codes. This output is passed through a data network to other subsystems. If the incoming data stream is incomplete or unstable, a process can augment it by accessing or buffering the data stream via the image record. However, only one transformation in a given processing configuration can be buffered at any moment. Coherent data streams (see [3]) may be blended at the input array of a subsystem, with the result that a process can 'engage' and transform data streams derived from multiple input sources.
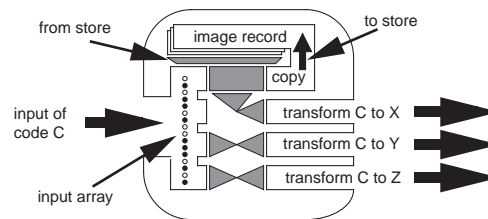

Figure 1: Generic structure of an ICS sub-system.

ICS assumes the existence of 9 distinct subsystems, each based on the common architecture described above:

Sensory subsystems
VIS   visual: hue, contour etc. from the eyes
AC   acoustic: pitch, rhythm etc. from the ears
BS   body-state: proprioceptive feedback

Meaning subs ystems
PROP   propositional: semantic relationships
IMPLIC   implicational: holistic meaning

Structural subsystems
OBJ   object: mental imagery, shapes, etc.
MPL   morphonolexical: words, lexical forms

Effector subsystems
ART   articulatory: subvocal rehearsal, speech
LIM   limb: motion of limbs, eyes, etc

Overall behaviour of the cognitive system is constrained by the possible transformations and by several principles of processing. Visual information for instance cannot be translated directly into propositional code, but must be processed via the object system that addresses spatial structure. Although in principle all processes are continuously trying to generate code, only some of the processes will generate stable output that is relevant to a given task. This collection of processes is called a *configuration*. As an example the thick lines in Fig. 2 show the configuration of resources deployed while using a hand-controlled input device to operate on some object within a visual scene. The propositional subsystem (1) is buffering information about the required actions through its image record and using a transformation (written :prop-obj:) to convert propositional information into an object-level representation. This is passed over the data network (2), and used to control the hand through :obj-lim: (3) and :lim-hand: (4) transformations. However, both obj and lim are also receiving information from other systems. The users' view of the rendered scene arriving at the visual system (5) is translated into object code that gives

a structural description of the scene; if this is to be blended at obj with the users' propositional awareness of their hand position (from 2) the two descriptions must be coherent. A propositional representation of the scene is generated by :obj-prop: and passed to prop (6) where it can be used to make decisions about the actions that are appropriate in the current situation. In parallel with this 'primary' configuration, proprioceptive feedback from the hand is converted by the body-state system (7) into 'lim' code (8) in a secondary configuration.
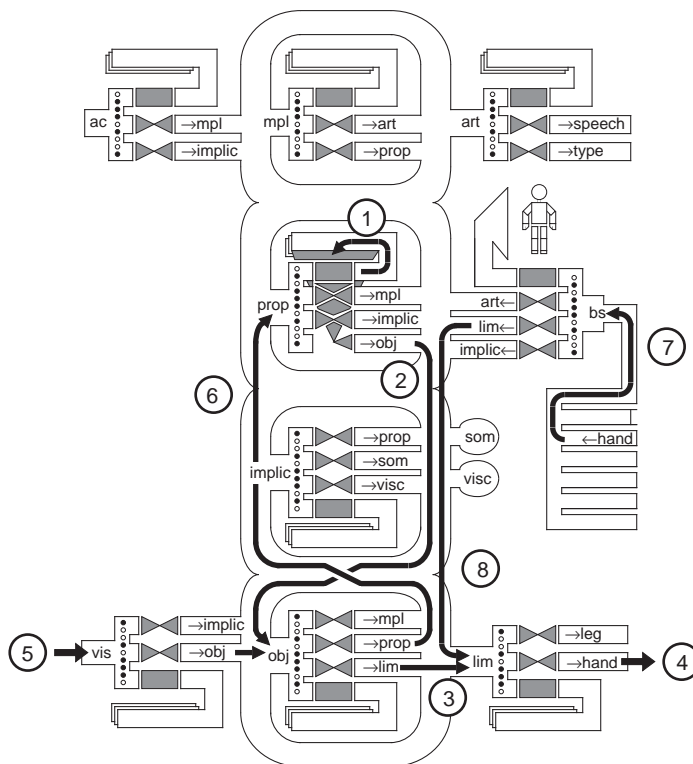

Figure 2: ICS (configured for graphical interaction).

The described configuration has been studied and formalized in [10] by using the Modal Action Logic (MAL) notation [19]. Briefly, MAL is a typed first-order logic that extends the predicate logic with an additional operator. For any action 'A' and predicate 'P', the predicate '[A]P' means that after the action A is performed, P must hold. The formal specification provides a novel insight into the usability of interaction devices. It shows that the relation between coordinate spaces in which an operator manipulates a device plays a significant role in the comparison of the ease of use of input devices.

In the sequel we study a similar configuration, but without the feedback from the hand to the 'lim' subsystem.

The key observation in this paper is that the structures and principles embodied within ICS can be formulated as a data flow oriented specification where the transformations performed by the subsystems can be modelled by continuous stream processing functions. This specification approach is also used for general information processing systems and therefore the descriptions of the ICS and a system could

be combined supporting the syndetic modelling approach. This means that the cognitive resources of a user can be expressed in the same framework as the behaviour of computer-based interface, allowing the models to be integrated directly. The data flow approach reveals another aspect of the use of input devices, namely the synchronisation problems that are caused by internal delay and feedback within the cognitive information processing process of the human.

3   The Functional Data Flow Approach

In the *functional data flow* approach [15, 6, 18] the behaviour of systems is essentially modelled as a number of independently operating subsystems that cooperate together by means of the exchange of messages.

The graphical representation of the system is a graph with nodes and directed arcs. The nodes represent the subsystems, the arcs the flow of information between these subsystems. Subsystems in their turn can again be described using a data flow approach.

The behaviour of the subsystems is modelled by functions on the incoming data streams that are represented by the directed arcs. Since these functions are modelling the sequential processing of data they must have two important properties. They never need an infinite amount of input before they produce any output and moreover once any output has been produced it can no longer be replaced by other output or deleted. So, when more input is supplied this can only lead to more output produced by the function. Functions on streams that enjoy these two properties are continuous. However, in this way only *deterministic* behaviour can be specified, i.e. the same input leads always to the same output. An extension to the description of non-deterministic behaviour is described in [18]. This extension may be very useful to model some aspects of ICS. For example if we want to model the variety in users reactions in apparently equal situations. Modeling these aspects is however out of the scope of this paper but an interesting topic of future research.

In this paper we use the functional programming language Miranda[2] for the definition of the stream functions. The way these functions are defined is rather close to how functions are defined in mathematics. One additional feature is that we can use pattern matching on the arguments of a function for an easy definition of different cases. A extended introduction to functional programming in a language similar to Miranda can be found in [7], a description of the Miranda language itself can be found in [20].

In this section we give only an example of some elementary features and we explain other aspects when we use them.

Functions in Miranda consist of two parts; a function *type* and a function *mapping*. The type part is optional, but we will include it because it gives additional information on the function on an abstract level that increases readability. For example, the function `plusfive`, that takes a number and gives as result the number increased by 5, can be defined as follows.

```
plusfive :: num -> num
plusfive n = n + 5
```

The first line gives the type of the function; it transforms natural numbers into natural numbers indicated by the arrow (`->`) type constructor and its parameters (`num`). The second line gives the function mapping where `n` stands for the formal parameter. We can use the function by applying it to any object of the right type, i.e. any number in this case. So, for example, `plusfive 6`. What happens is that

---

[2]Miranda is a trademark of Research Software Limited

by *pattern matching* the formal parameter `n` is associated with the number 6 and thus in the expression `5 + n` the `n` stays for 6, resulting in `5 + 6 = 11` as result of `plusfive 6`.

In Miranda, type constructors allow to compose more complicated types from basic ones (`num`, `bool`, `char`). For example in this paper we frequently use the type constructor for lists (sequences). The type "list of numbers" is denoted as `[num]`. A function that transforms lists of numbers into lists of characters then has type `[num] -> [char]`. We can introduce new names as shorthands for types like `numlist=[num]`.

The notation for one particular list is quite similar to that of the type of lists. If we want to denote the list with the three numbers 1, 2, 3 it is written as `[1,2,3]` in Miranda. The list that does not contain any number is written as `[]`.

Formal names, such as `n` in the function `plusfive` can also be used to denote lists. Often it is useful to have separate names for the first element of a list and for the rest of the list specially for pattern matching in recusively defined functions of which we will give an example later on. A list with first element called `a` and the rest called `as` can be denoted as `a : as`, pronounce "a prefix as". We can use always brackets to denote grouping of components, so `(a : as)` denotes the same pattern. Actually the colon is a so called "list combinator"; an operator that combines one element with a list in order to get one new list where the first element is the element that was added to the list. All elements of a list must always have the same type. The list combinator can also be used on concrete objects, for example: `1 : [2,3,4]` gives as result the list `[1,2,3,4]`. The use of the list combinator on abstract names is very useful for pattern matching. For example the function that calculates the length of a list of items of any type (denoted by `*`) can be defined recursively as follows:

```
length :: [*] -> num
length [] = 0
length (a : as) = 1 + (length as)
```

It says that the length of an empty list is zero and that the length of a list `(a:as)` is equal to one plus the length of the list without the first element. If we apply `length` to the list `[1,2]`, then `a` is associated with 1 and `as` with the rest of the list `[2]`. So we get the following evaluation:

```
length [1,2] =   {apply definition}
1 + length [2] =   {apply def. again}
1 + 1 + length [] =   {def.}
1 + 1 + 0 = 2 .
```

Pattern matching is not the only way in Miranda to discriminate between different cases. The same length function can also be defined using the general conditional construction.

```
length :: [*] -> num
length s = 0 , s = []
        = 1 + (length (tail s)) , otherwise
```

In this definition `tail s` is the list `s` without the first element. The choice which case selection technique is used depends on the kind of conditions that have to be formulated for the different cases. A program in Miranda (script) consists of a collection of function definitions and expressions that are

composed of functions applied to their actual arguments. Functions can be composed by function composition (`.`) defined as `(f.g) x = f (g x)`. The functions can be evaluated within the interactive Miranda interpreter.

4   Synchronisation and Feedback

In this section we first describe how some of the ICS concepts can be modelled in the functional data flow approach, followed by a discussion of the issues concerning synchronisation and delay due to feedback. In order to clearly illustrate the main issue in this article, we chose to model the subsystems and the flow of information as simple as possible. We are aware that this does not reflect properly the full ICS model but it serves well for concentrating on the issue at hand. In other papers we will elaborate on other aspects of the ICS model and integrate the various concepts into a more accurate model.

4.1   Modelling the Information Flow

From the informal description of the ICS network in Section 2 we can observe nine different subsystems that communicate with eachother by the exchange of mental representations. Each subsystem is trying to produce information continuously, but only some transformations will succeed in producing stable information streams that then become part of a particular stable configuration. This is the starting point for our model of the flow of information within the network. This concept can be modelled quite naturally by what is called a *broadcast architecture* of information distribution.

The basic principle is that every subsystem sends messages to other subsystems via a common communication network represented as a separate component in the model. The messages exchanged between the subsystems contain the name of the subsystem it is addressed to and the mental representation that is communicated to that subsystem. The network component receives all the messages that are sent by the subsystems and distributes them to all the subsystems. This means that every subsystem receives all the information of all other subsystems, but since each message contains the name of the subsystem to which it is addressed, each subsystem can select exactly the information that was addressed to it.

This way of modelling the communication structure was chosen in order to have a static framework within which different configurations can be studied. Note that the observations of a person of the environment (input) and his reactions to the environment (output) are put directly on the data network instead of on the specific sensor/effector systems. This way the network serves for the transfer of all information and as an interface to the outside world. The input from the environment is then directed to the proper subsystem to handle it.

The transformations that take place within the subsystems are modelled by continuous stream processing functions that transform streams of messages. So, for example the transformation (written `:prop-obj:`) in the propositional subsystem that converts propositional information into an object level representation can be modelled by a function that takes a stream of (a representation of) propositional information and produces the corresponding stream of object level representations. We will denote such transformation functions by a name similar to the name of the transformation in the informal description of the ICS model. For example the transformation `:prop-obj:` will be denoted by the name `prop_obj`. In a more detailed model we can discriminate separate parts within each subsystem that take care of engagement in a stream, blending of streams and the use of a buffered stream. However, in this article we concentrate on delay and synchronization aspects that are related to reciprocal loops in configurations and their relation to buffered information processing.
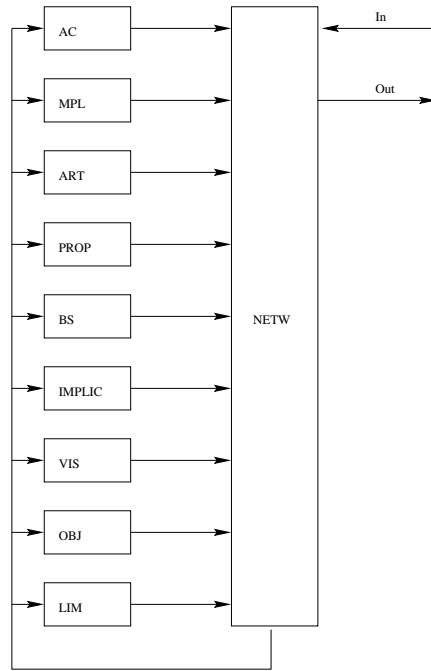
Figure 3: A broadcast based architecture of the ICS model

To illustrate the problem at hand we take a very simple configuration as an example throughout the paper. In this configuration only four subsystems are involved. The configuration denotes a person that looks at his hand that he moves with increasing velocity. The feedback the person observes from his hand is considered to be only visual. In other words, he disregards the feedback that could be obtained via observing the body state. Further we assume that the person moves his hand consciously with a certain speed, so that the propositional subsystem is involved. So, in this configuration the visual, the object, the propositional and the lim subsytems are involved. For keeping the example clear we consider a reduced model that contains only the four mentioned subsystems, the hand and the network component. We give a formal description by using the functional programming language Miranda. The particular choice for Miranda is not essential. The same approach could be taken using any functional language.

To keep the link to the ICS model clear, we present first in a schematic way the configuration under study in Fig. 4.
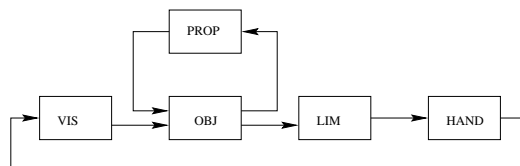

Figure 4: Configuration for moving one's hand

The corresponding formalization in a broadcast oriented structure can be depicted again as in Fig. 3. However, many of the subsystems there would not produce stable information because they are not part of the configuration. Therfore, without loss of generality, we study a reduced broadcast structure involving only the relevant subsystems as presented in Fig. 5.
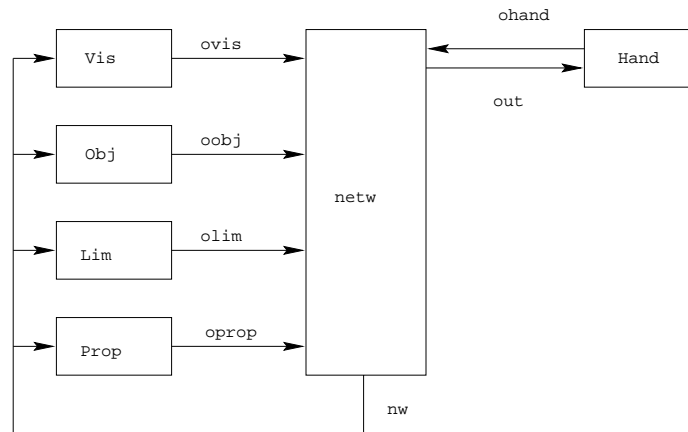

Figure 5: Broadcast structure for moving one's hand

The information that is exchanged between the subsystems is kept extremely simple. They are just messages consisting of two parts. The first part contains the name of the subsystem to which it is addressed. The second part the value that models the mental representation that is communicated. In this example it is denoted by a plain number. In Miranda the type of the messages can be defined in the following way. First we define the names of the subsystems (components) to which we can address messages. This way we define a new type of objects consisting of the names of the involved subsystems.

```
subsys ::= Vis | Obj | Prop | Lim | Hand
```

Then we define the type of messages as pairs of address (target) and mental representation (val). The target is the name of one of the subsystems and the mental representation is modelled by numbers to keep the example simple. Of course more complicated types to model mental representations may be necessary in other configurations.

```
msg == (target, val)

target == subsys

val ==  num
```

In order to access the individual parts of a message we define two selector functions that, given a message, produce the selected part. The function **tgt** gives the address part of the message, and the function **nva** gives the value part of a message. In Miranda we can use pattern matching to introduce names for the two parts of a message (**t** and **v**).

```
tgt :: msg -> subsys
tgt (t,v) = t

nva :: msg -> num
nva (t,v) = v
```

The third useful general function we need is a selection function that can filter out those messages from a list of messages that are addressed to a specific subsystem. The function `filter` is standard in Miranda and takes a filtering predicate and a list and gives as result a list of elements that satisfy the predicate. In the selection function the filtering predicate is composed of two functions by means of function composition (denoted by . in Miranda). First the function `tgt` selects the address of the message and then the function `=sysname` checks whether this is equal to the name `sysname`. If so, it is selected from the list `ms` and the next element of the list is checked.

```
sel :: subsys -> [msg] -> [msg]
sel sysname ms = filter ((=sysname).(tgt)) ms
```

For example `sel Obj [(Vis,2),(Obj,6),(Obj,4)]` gives as result `[(Obj,6),(Obj,4)]`.

The overall behaviour of the network can be captured by a set of equations that can directly be derived from the Data Flow Graph of Fig. 5. This gives us a definition that reflects mainly the interconnection structure of the various components. The information exchange is modelled in a synchronous way; each subsystem has to produce a message in every step and sends it to the network (`netw`). The network in its turn receives a message from every subsystem in each step. In fact it has as arguments the lists produced by each component. It produces as a result a list of messages [`msg`] in each step which is the collection of received messages from each component. This it sends to every subsystem. In the following part of the Miranda model `ovis` is the produced list of messages by the subsystem `vis`, `oobj` that of subsystem `obj`, `olim` that of `lim` and `oprop` that of `prop`. The names `vis`, `obj`, `lim` and `prop` stand for the functions that model the behaviour of the subsystems with the corresponding names respectively.

```
ovis :: [msg]
ovis = (Obj, 0) : ( vis nw )

oobj :: [(msg,msg)]
oobj = ((Prop,0),(Lim, 0)) : (obj nw)

olim :: [msg]
olim = (Hand, 0) : (lim nw)

oprop :: [msg]
oprop = (Obj, 0) : (prop nw)

ohand :: [msg]
ohand = (Vis,0) : (hand out)

nw :: [[msg]]
nw = netw ovis oobj olim oprop ohand
```

The network function can be defined as follows. In every step it takes one message from each stream coming from a subsystem and packs them into a small list `[a,b1,b2,c,d,e]`.

```
netw :: [msg] -> [(msg,msg)] -> [msg] -> [msg] -> [msg] -> [[msg]]
netw (a : as) ((b1,b2):bs) (c:cs) (d:ds) (e:es)
  = ([a,b1,b2,c,d,e]) : (netw as bs cs ds es)
```

The behaviour of the various subsystems can be modelled by continuous functions that transform streams of messages. The visual subsystem receives information from the environment. In this example it observes a very specific kind of information which is the speed of the hand movement. The subsystem Vis is simply modelled as a function `vis` that in every step selects from the incoming messages `ms` those that are addressed to Vis (`(sel Vis ms)`), which in this example gives a list with one message. To get the message itself we use list indexing which in Miranda is denoted by `!`. The first element of a list has index `0`. The value of that message is obtained by `nva` (i.e. number value) and it is sent to the object subsystem as (`Obj,value`). The subsystem `vis` then proceeds with the next step recursively (`vis mss`).

```
vis :: [[msg]] -> [msg]
vis (ms : mss) = (Obj, value ) : ( vis mss)
                  where value =  (nva ((sel Vis ms)!0))
```

The subsystems Lim and Prop are modelled in a similar way:

```
lim :: [[msg]] -> [msg]
lim (ms : mss) = (Hand, value) : (lim mss)
                  where value = (nva ((sel Lim ms)!0))


prop :: [[msg]] -> [msg]
prop (ms : mss) =  (Obj, value) : (prop mss)
                    where value =  nva ((sel Prop ms)!0)
```

For this example only one transformation function in each of the subsystems Vis, Lim and Prop are part of the configuration under study. The above functions actually are the definitions of these transformation functions. So `vis` is `vis-obj`, `lim` is `lim-hand` and `prop` is `prop-obj`.

The object subsystem is a bit more complicated in this example because it must be modelled by two parallel internal transformation functions. Therefore we refine this subsystem be defining the internal transformation functions explicitely. Each transformation function, `obj_prop` and `obj_lim` produces its own list of messages. These two lists are combined into a list of pairs that can be offered to the network. This is performed by the standard function `zip2` in Miranda that does exactly this job.

```
obj :: [[msg]] -> [(msg,msg)]
obj mss = zip2 (obj_prop mss) (obj_lim mss)
```

The function `obj_prop` receives all messages from the network, selects those that are addressed to `Obj` by (`sel Obj ms`) and that come from the Visual subsystem (index 0, i.e. (`!0`)). It sends messages to the propositional subsystem ((`Prop,value`)). So it sends actually just the value it received directly from the visual subsystem.

```
obj_prop :: [[msg]] -> [msg]
obj_prop (ms:mss) = (Prop, observed_value): (obj_prop mss)
                    where observed_value = (nva ((sel Obj ms)!0))
```

The function `obj_lim` also receives all messages from the network, selects those addressed to it and compares the values that arrive from subsystem Prop and subsystem Vis. If these values are different, then the value 0 is sent to the Lim subsystem. If, however, the values are the same, it instructs the Lim subsystem to increase the velocity of the hand by one.

```
obj_lim :: [[msg]] -> [msg]
obj_lim  (ms:mss) = (Lim,0) : (obj_lim mss), different fromvis fromprop
                  = (Lim,(fromvis +1 )) : (obj_lim mss), otherwise
                    where different a b = (a ~= b)
                          fromvis = nva ((sel Obj ms)!0)
                          fromprop = nva ((sel Obj ms)!1)
```

This could be interpreted as that the speed of the hand is increased when the person gets aware of the actual speed. In the time that the information on the actual speed did not yet pass through the propositional subsystem, the Lim subsystem is temporarily producing somehow unstable information, denoted by the 0 in order to easily identify these points in the results that we will discuss later.

In this example the hand of the person is actually considered as part of the environment. In more elaborated examples we will of course consider a computing system or an interface as the object with which a user is working. Here we want to keep things as simple as possible however. Therefore we define the behaviour of the hand as follows:

```
hand :: [msg] -> [msg]
hand (m:ms) =  (Vis, value): (hand ms)
               where value = (nva m)
```

The hand takes as input those messages of the network that are addressed to it. These are those considered as output. This is performed by the standard function `map` in Miranda, that applies a functions to each element of a list. In this case it selects from the stream of lists of messages all those addressed to the Hand by `((!0).(sel Hand))`.

```
out =  (map ((!0).(sel Hand)) nw )
```

At this point we have a complete functional specification of this configuration in Miranda. The specification can be "run" as a program interactively within the Miranda environment. By typing for example "ovis" we can observe all the messages that are produced by the visual subsystem. By typing other names of channels we can observe also the messages passing there. Of course for obtaining a readable layout of the output we should use a printing function that takes care of the layout of the output. These can also be defined in an easy way in Miranda, but we did not include them in this article.

4.2   Analysing the configuration

When we indeed "run" the program and we observe the messages that are sent around, we can observe that they do not establish the result we would have liked to obtain. We would have liked to see that the speed of the hand would increase step by step. However, we get the following result when we observe only the values of the messages:

```
Step:     ovis   oobj    olim oprop ohand    out
  1:        0     0     0    0    0    0       0
  2:        0     0     1    0    0    0       0
  3:        0     0     1    1    0    0       1
  4:        0     0     1    1    0    1       1
  5:        1     0     1    1    0    1       1
  6:        1     1     0    1    0    1       1
  7:        1     1     0    0    1    1       0
  8:        1     1     2    0    1    0       0
  9:        0     1     2    2    1    0       2
 10:        0     0     0    2    1    2       2
 11:        2     0     0    0    0    2       0
 12:        2     2     0    0    0    0       0
 13:        0     2     0    0    2    0       0
 14:        0     0     0    0    2    0       0
 15:        0     0     0    0    0    0       0
 16:        0     0     1    0    0    0       0
 17:        0     0     1    1    0    0       1
 18:        .     .     .    .    .    .       .
 19:        .     .     .    .    .    .       .
 20:        .     .     .    .    .    .       .
```

From this result we see immediately that the model is not much in agreement with what we would like to obtain, namely an increasing hand speed. But what is exactly the problem ? Let's take a careful look at the results.

Initially (step 1) all the subsystems, including the hand, produce an initial value. This value has been put to 0, corresponding to the fact that initially the hand does not move. In this situation the values that the object subsystem receives from VIS and PROP are equal, and this means in this example that OBJ instructs LIM to start moving the hand. It does this by sending the value 1 (speed 1 let's say), see step 2 at the second column of `oobj`. A look at Fig. 5 helps in keeping track of the flow of the values.

In the following steps 3 and 4 we see that the value 1 is propagated to LIM and to the hand and that the hand indeed increases its speed. Note that meanwhile (step 3 and 4 again) the subsystem VIS could not yet observe an increase in the speed of the hand, so that OBJ is repeating the same value a couple of times to the propositional subsystem (first column of `oobj`). In step 5 however, we see that VIS noticed the increased speed of the hand and informs OBJ of this fact. PROP, however, is still receiving the value 0 from OBJ, and is actually two steps behind with repect to what VIS has been observing. This results in that OBJ is producing two zero's, which symbolises the fact that PROP and VIS are not yet working with the same information, or, in other words, the information OBJ gets from VIS and PROP are not yet concerning the same observation and are thus not yet synchronised.

So, OBJ sends two zeros that represent the fact that PROP and VIS are out of synchronisation, and in the meantime informs PROP about the newly observed speed (steps 6 and 7, first column of `oobj`).

After two steps the information OBJ gets from VIS and PROP are again synchronised (both are 1 now, see step 7) and thus, in step 8 OBJ can again increase the speed of the hand which then becomes two. However, observe that with one step delay in step 7 and 8, the speed of the hand dropped temporarily to zero, due to the fact that there was a delay in the information update of the person.

Of course, in reality, the speed of the hand would not drop to zero for a while. In this very case we could have modelled the behaviour of OBJ in such a way that when the incoming information is not synchronized, the speed is kept unchanged during that period. This would lead to the results we were expecting and thus solve the problem. That this solution works in this case is due to the fact that the hand movement is completely under control of the person who is moving it. Interesting is that with this solution the output sent to the hand is always repeated during a certain period of time. This happens because an observation at VIS is transferred (and transformed) a number of times through the network from one subsystem to another before it "appears as output". This may be an interesting aspect related to the fact that from psychological experiments it is known that there is a tendency that certain stable streams of information are sustaining for a while.

However, in general the environment of a person may change fast and out of the direct control of the person, and in that case sustaining information for a while does not help solving synchronisation problems.

4.3   Sampling and diversity in relative processing speed

Another way to look at the same problem of synchronization is to consider it as natural that there are periods in which information that has to be processed by different subsystems before being fed back into another subsystem is unstable. In computer science this fenomenon of instability is certainly not uncommon. The "bits" in a wire, corresponding to high and low voltage are an example of this. For a current to go from high to low voltage (and vise versa) it takes a small amount of time. In order to observe a bit (zero or one) we have to observe the wire at the right moment (at a stable maximum or minimum). This "watching at the right moment" is called *sampling*. A similar technique can also be applied to our example. This means that the hand takes note of the outcome of the LIM subsystem only once in a while and in this way is not bothered by fluctuations due to instability, desynchronization or for example delay due to initial values. In order to let the visual subsystem observe the speed of the hand for a relatively longer time, we can repeat the value that can be observed from the hand a number of times. At the output of the LIM subsystem we can then sample the information stream with the same rate. The points where we sample and repeat information should, in a more realistic setting, be on the border between the user and the system. In our example we will insert them around the hand as is shown in Fig. 6.
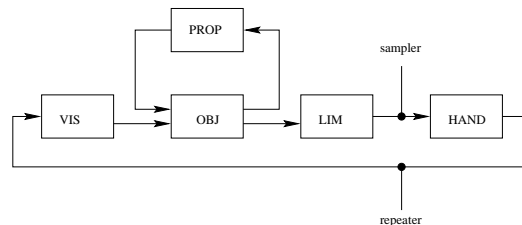

Figure 6: Configuration with repeat and sampling points

In the broadcast architecture these repeat and sampling points are to be inserted at the input and output channels of the model.

A function that repeats each message a number of times, and a sampler can be easily defined in

Miranda. The repeater takes a number, **n**, for the number of times it has to repeat each message, and a list of messages. It gives as the result a list of messages where each one is repeated **n** times.

```
repeater :: num -> [*] -> [*]
repeater n (m:ms) = (rep n m) ++ (repeater n ms)
```

The sampler does just the opposite of the repeater. It takes the sampling rate and a list of messages and gives the sampled list as a result. Note that it everytime takes **n** messages from the list, and keeps only the last of these selected messages. This way the sampling function can also be used to skip the initially produced output which are a result of the initialisation values in the network and thus unrelated to any observed input. In the definition below **take** and **drop** are standard Miranda functions that respectively give the first **n** elements of a list **ms** and give the list without the first **n** elements.

```
sample :: num -> [msg] -> [msg]
sample n ms = (take n ms)!(n-1) :  (sample n (drop n ms))
```

In order to insert the repeater and the sampler in the network we change the program a bit. We do it in such a way that we can observe again the values that are passing through the network.

First of all we introduce a constant denoting the rate of repeating and sampling and thus the relative speed between the user processing information and the changes in the by the user observed environment (which in this example happens to be the hand).

```
rate = 6
```

The rate can be changed of course and we will investigate the results for different rates.

We change the definition of the output of the hand as follows:

```
ohand ::  [msg]
ohand = repeater rate ((Vis, 0) : ohandi )

ohandi  = hand osampler
osampler  = sample rate out
```

With sampling and repetition rate equal to **6** we now obtain the following result:

step:	ovis	oobj	olim	oprop	hand	osampler	rate = 6
1:	0	0	0	0	0	0	
2:	0	0	1	0	0	0	
3:	0	0	1	1	0	0	
4:	0	0	1	1	0	0	
5:	0	0	1	1	0	0	
6:	0	0	1	1	0	0	1
7:	0	0	1	1	0	1	
8:	1	0	1	1	0	1	
9:	1	1	0	1	0	1	
11:	1	1	0	0	1	1	
12:	1	1	2	0	1	1	
13:	1	1	2	2	1	1	2
14:	1	1	2	2	1	2	
15:	2	1	2	2	1	2	
16:	2	2	0	2	1	2	
17:	2	2	0	0	2	2	
18:	2	2	3	0	2	2	
19:	2	2	3	3	2	2	3
21:	2	2	3	3	2	3	
22:	3	2	3	3	2	3	
23:	3	3	0	3	2	3	
24:	3	3	0	0	3	3	
25:	3	3	4	0	3	3	
26:	3	3	4	4	3	3	4

From this result we see immediately that the input to the hand, represented in the column of `osampler` is now indeed increasing with increment one all the time, which represents the increase of the intended speed of the hand. It is also clear that the information on the speed of the hand is repeated 6 times after every change. In fact in this way the human has **6** time slots (processing cycles) to reach a stable output. The sampling function samples the produced output by LIM indeed exactly after it became stable again (compare steps 6, 13, 19 etc of column `olim` and `osampler`).

Note that if we would reduce the rate to 1, we obtain exactly the same results as in the previous case. The minimum rate that will guarantee a correct result depends on a number of aspects like the delay caused by internal feedback loops in a configuration and the delay caused by functions that model the behaviour of the subsystems. In our example the last kind of delay did not occur because every function produces its output only based on the actual input (history insensitive). The first kind of delay appeared in the feedback loop between OBJ and PROP.

Note that the sampling technique solves the problem of synchronization in a more general way than just by sustaining a value during instable periods. It may be used to clarify the relative speeds between the user processing information on the observed environment and the speed of the changes in the environment itself. If the environment changes too fast, i.e. if the rate is too small, the user cannot react appropriately to each change and we will observe instable output.

If we would take any rate higher than the minimum, the same results as for the minimum are obtained. In fact the relative time during which the instability occurs gets shorter when the rate is higher. However,

in reality we can expect that this does not mean that a very high rate leads to a better performance of information processing of the human. Of course with a greater difference between human processing speed and the pace in which the environment changes, the human perceives the changes in environment as slow and might loose concentration or gets bored or distracted. A higher rate will therefore in reality not always lead to better performance. The data flow model, as it is now in its very simple form, does not give a maximum rate however. Further enhancement of the model, with concepts as engagement and disengagement and stability of streams may lead to a model that also captures the derivation of maximal rates. This is an interesting topic of further research.

5   Feedback and buffered processing

So far we have been mainly concerned with delay caused by feedback of information via the network. Another cause of delay is the internal processing of a subsystem. In this section we investigate their relation.

In this paper the ICS theory has been modelled by means of a synchronous (broadcast) architecture. This means that the network in every cycle (step) collects a message from each of the subsystems (and the in and output) and distributes them. This way the time needed for a cycle is determined by the time it takes for the slowest subsystem to transform its input message into output to the network. From the ICS theory it is known that processes that transform information in buffered mode are significantly slower than those that can transform the information directly. The fact that the slowest process determines the speed of the overall configuration thus corresponds to what happens in the data flow model.

However, from the theory on ICS we know also that there may be more than one indepent configurations active at a time. For example we can speak and walk at the same time, which requires different configurations to be active. In a model as in Figure 3 this would mean that the speed of one configuration could influence the speed of the other, since the time for a cycle is depending on the slowest process in any of the configurations in this model.

This might however not be the appropriate way to model two simultanuously active configurations. An alternative may be that a slow subsystem, that operates in buffered mode, produces intermediate "empty" messages so that the network can proceed with the next cycle. This way only the progress of the subsystems within the configuration that contains the buffered subsystem are influenced by the delay due to the insertion of the empty messages. Which of the two ways to model this issue is most appropriate is a topic of further research.

6   Conclusions and further research

In this paper we proposed a data flow oriented approach to model particular aspects of the ICS model that are related to synchronization and delay. We have shown that both the internal delay of a subsystem and the feedback of information through the subsystems (reciprocal loops) can lead to delay in producing stable output. Feedback may moreover cause the production of instable output because the information that reaches the subsystem may arrive in a non-synchronized way, causing distortion in the transformation.

One way to deal with this problem is to create a difference in the pace in which information at the sensors changes and the pace with which the cognitive model of subsystems processes the information. In a data flow approach this difference can easily be established by repeating the input messages that go

to the ICS model and by sampling the output that comes from the model. The number of repetitions depend on the particular structure of the configurations. Essentially it depends on how many cycles of the network are necessary before the to the input related output appears as a result of the network.

Since there are experimentally derived results available about the average time needed for a subsystem to transform information, both in buffered and in direct mode, the model may be used to get an indication of the time aspects of cognitive configurations and the pace of changes in the environment a user can cope with. For example it is known that subsystem need approximately 40 ms to process mental representations, this model might therefore also be interesting for reasoning about absolute timing aspects.

The example in this article also shows how a formal approach to describing a user model of cognition can raise new questions about the theory that can eventually improve the understanding of human cognition.

As part of further research other aspects of the ICS model can be added to the data flow model such as blending, engaging and disengaging and the change from one configuration to another.

The translation of the model into a specification in Promela [14], the specification language that is used in the model checker SPIN [14], may result in the possibility to check automatically properties of a combined specification of both cognitive user aspects and system aspects. This would be another step towards the syndetic modelling paradigm that has been proposed and developed within the AMODEUS project.

References

[1] R.M. Baecker and W. Buxton, editors. *Readings in human-computer interaction: A multidisciplinary approach.* Morgan-Kaufmann, 1987.

[2] P.J. Barnard and J. May. Cognitive modelling for user requirements. In P.F. Byerley, P.J. Barnard, and J. May, editors, *Computers, Communication and Usability: Design Issues, Research and Methods for Integrated Services*, North Holland Series in Telecommunication. Elsevier, 1993.

[3] P.J. Barnard and J. May. Interactions with advanced graphical interfaces and the deployment of latent human knowledge. In *Eurographics Workshop on Design, Specification and Verification of Interactive Systems*, pages 15–49. Springer, June 1994.

[4] A. Blandford and D.J. Duke. Integrating user and computer system concerns in the design of interactive systems. *IEEE Transactions on Software Engineering*, 1996. Submitted for publication.

[5] R. Boute. An introduction to funmath. Introductory notes for the course 'Basic Mathematical Complements for Computer Science' - University of Gent, Belgium, 1994.

[6] M. Broy. Compositional refinement of interactive systems. In *Program Design Calculi - International Summer School* [16].

[7] R. Bird and Ph. Wadler. *Introduction to Functional Programming.* Prentice-Hall, 1988.

[8] S. Card, J. Mackinlay, and G. Robertson. The design space of input devices. In *Proc. of CHI'90.* ACM Press, 1990.

[9] S.K. Card, J.D. Mackinlay, and G.G. Robertson. A semantics analysis of the design space of input devices. *Human-Computer Interaction*, 1990.

[10] D.J. Duke and G. Faconti. Device Models. *To appear in the proceedings of the DSV-IS96 conference*, Namur, Belgium, 1996.

[11] D.J. Duke, P.J. Barnard, D.A. Duce, and J. May. Systematic development of the human interface. In *APSEC'95: Second Asia-Pacific Software Engineering Conference*, pages 313–321. IEEE Computer Society Press, 1995.

[12] D.J. Duke and M.D. Harrison. Interaction and task requirements. In P. Palanque and R. Bastide, editors, *DSV-IS'95: Eurographics Workshop on Design, Specification and Verification of Interactive Systems*, pages 54–75. Springer-Verlag, 1995.

[13] P.M. Fitts. The information capacity of the human motor system in controlling amplitude of movement. *Journal of Experimental Psychology*, 47:381–391, 1954.

[14] Holzmann, G. *Design and Validation of Computer Protocols*. Prentice Hall, 1991, ISBN0-13-539925-4.

[15] G. Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the 1974 IFIP Congress*. IFIP, 1974.

[16] Program design calculi - international summer school, 1992.

[17] P. Chan J.D. Foley, V.L. Wallace. The human factors of computer graphics interaction techniques. *Computer Graphics and Applications*, 4(11), 1984.

[18] M. Massink. *Functional Techniques in Concurrency*. PhD thesis, University of Nijmegen, February 1996. ISBN 90-9008940-3.

[19] M. Ryan, J. Fiadeiro, and T. Maibaum. Sharing actions and attributes in modal action logic. In T. Ito and A.R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 569–593. Springer-Verlag, 1991.

[20] D. Turner. *Miranda System Manual* Research Software Limited, 1987.