A LOTOS specification of a CSCW tool

J. Rekers *I. Sprinkhuizen-Kuyper*

Department of Computer Science, Leiden University
P.O.Box 9512, 2300 RA Leiden, The Netherlands
Email: rekers@rulwi.leidenuniv.nl, phone: +31 71 277092, fax: +31 71 276985

ABSTRACT

CSCW systems provide computer mediated support for the collaboration of people working in a group. We investigate whether a specification formalism for communicating processes can be used to describe the communicative aspects of these systems. Such a specification should make the communication patterns evident and should allow for automatic generation of a computer program that implements it.

In this paper we apply this approach on a simple example: we develop a LOTOS specification of the CSCW tool We-Met and check whether the resulting specification meets the objectives above.

1 INTRODUCTION

Tools which provide high level support for the communication between members of a group need to be tailored towards the specific needs of that group. It is the aim of the MOCCA group (Modeling Of Coordinated Collaborative Activities) at Leiden University to develop a specification language to define the communicative needs of groups. It should be possible to generate the corresponding CSCW (Computer Supported Cooperative Work) tool from such a specification.

The focus of this specification language will be on communication and the language should allow for mathematical reasoning about it. We intend to look at the members of a group as a set of cooperating processes which communicate through a certain protocol. Such a protocol might range from highly structured (as will be the case with form processing in office environments) to offering no structure at all (as would be more appropriate for design discussions).

1.1 A classification of CSCW tools

In order to specify the communicative aspects of CSCW tools, we need an overview of the existing tools first. We consider the following notions important for such a specification and will use these to classify CSCW tools with: 1) does the tool offer synchronous or asynchronous communication, and 2) does it structure the communication or not. We explain these notions and give some examples.

synchronous communication If a tool offers synchronous communication, everything a participant utters will available to the other participants immediately. Messages are not send or received explicitly. The chunks of information will in this case generally be small and there will be something like a common data object which is edited concurrently by all participants.

asynchronous communication In tools which offer asynchronous communication participants prepare messages in isolation, and, when finished, they send it out explicitly. Receivers of a message are notified and they perform an explicit action to receive the message. The chunks of information will generally be larger. This kind of communication is less appropriate for editing a common data object jointly.

structured communication There are cases in which the communication among the members of a group follows clear and predefined paths. An example is the routing of a purchase request through an organization. Tools which support this kind of communication know these paths and will perform the routing automatically. These tools structure the communication.

unstructured communication Communication as it occurs in, for example, brainstorm sessions is inherently unstructured. Contributions are distributed to every participant and who react to what is unpredictable. Tools which support this kind of discussions only support the communication, they do not structure it. Unstructured communication and synchronous communication often go together.

We classify a few CSCW tools according to these notions.

We-Met (IBM – [27])

This is a pen-based tool which supports the communication and information retrieval needs of small group meetings. It is primary meant to support design discussions. If offers synchronous communication on broadcast basis and does not impose any structuring on the communication.

Object Lens, Information Lens (Xerox – [19, 22])

A mail based system with objects and agents. Objects are files which may contain references to other objects. Agents are user-defined and are triggered by events to execute their procedure on objects. A very flexible system, more concerned with the individual users than with the group as a whole. Asynchronous communication, amount of structuring depends on the agents applied.

ICICLE (Bellcore – [6])

ICICLE supports code inspection, which is a phase in the software development cycle between implementation and testing. This task is performed by a group of which the members have well defined tasks and roles. ICICLE has strongly structured procedures and applies both synchronous and asynchronous communication.

gIBIS (MCC - [9])

This tool supports the software design methodology Issue Based Information Systems (IBIS). The methodology is fixed but allows quite general communication. Although the communication is asynchronous (based on mail), all messages are composed in a single common design document, which is inspected by all participants.

Quilt (Bell – [20])

A hypertext system for collaborative writing and reviewing of papers. It supports the communication and information sharing among the collaborators on a document. The main structuring it puts on the communication are the privileges of who may read or write what information. Asynchronous communication.

X-Workflow (Olivetti – *refs?*)

This is a form routing system. The communication is structured by procedures which are written as Petri nets. These procedures are normally not adapted by users of the tool. Asynchronous communication.

1.2 Specification formalisms for communicating processes

There are two major directions in this field: Process algebra's and Nets.

There are several specification languages for concurrent, communicating systems which work according to the process algebra approach. These are all formal reasoning systems in which variables represent processes. Processes can be composed by operators like *sequence*, *alternative* and *parallel*, and the communication between processes is synchronous. The best known examples of this approach are:

- Communication Sequential Processes (CSP) [16]
- Calculus of Communicating Systems (CCS) [23]
- Language of Temporal Ordering Systems (LOTOS) [4]
- Algebra of Communicating Processes (ACP) [2]

Nets are extended finite state machines. The best known example of this approach is Petri Nets [24, 25]. Petri Nets represent a system by a directed bipartite graph of *channels* (or places) and *agencies* (or transitions). Channels passively store tokens. Agencies actively consume, transport, change and produce tokens. An agency may be activated if every of its input channels contains an token. The agency then consumes these tokens, processes them internally, and generates an token at each of its output channels. Communication is based on queueing and thus asynchronous.

It is often considered as a disadvantage that Petri Nets do not support any kind of modularity and are monolithic. In view of this several approaches have been proposed, like Statecharts [15] and Paradigm [14], aiming at, among other things, support of modularity. Both approaches are based upon a vector of finite state machines. Both are mainly graphical, although specifications can be purely represented by formulas. The communication between the various finite state machines in Statecharts is synchronous, whereas in Paradigm it can be synchronous or asynchronous.

If we compare the process algebra and the Nets, we see that process algebra's are quite formal. They allow for mathematical reasoning, but have as drawback that the specifications are hard to comprehend at a first glance. Net-like approaches have appealing graphical representations, but are quite often much harder to reason about.

1.3 An instance of the general idea

We verify our idea of specifying the communicative part of CSCW tools by writing an example specification of one of these tools. This means that we have to commit

ourselves to a specific formalism and to a single tool. It is already quite clear that CSCW tools which offer asynchronous communication and structure the communication in a rigid manner can be specified easily by petri net like approaches, as demonstrated by X-Workflow (Olivetti). We will therefore concentrate on the other side of the spectrum: on tools which are based on synchronous communication and do not structure the communication. We-Met is a good representative here. For these kind of tools it is plausible to apply a specification formalism which already offers synchronous communication. We choose here for LOTOS, as it even has multi-way synchronization. Furthermore, LOTOS is quite popular and a large number of supporting tools is available. LOTOS has already been used to specify large, real-life, systems. To mention a few application fields: several OSI layers, telephone systems [12, 5], power plants [3] and software process modeling [26]. The language is starting to be used in industrial environments also.

We understand that general conclusions cannot be drawn from this single example, but still, such a specification might give an indication of the feasibility of our ideas and provides the experience necessary to judge specification formalisms.

1.4 Overview of the paper

Section 2 and 3 contain respectively an overview of LOTOS and a description of We-Met. Having provided this basis, we develop in section 4 the actual LOTOS specification of We-Met. We propose some extensions to this specification in section 5 and in section 6 we evaluate our work and set some directions for further research.

2 LOTOS

We mention some of the main keywords of LOTOS here but we must refer to other sources for a real introduction: a good start is [21] which illustrates most features of the formalism with example specifications. [4] is an already more formal approach to LOTOS, but still easy to read. The complete description of LOTOS can be found in [17].

LOTOS consists of two clearly separated sub-formalisms. The part that deals with processes and their interaction is influenced by both CCS [23] and CSP [16]. The part that deals with the declaration of datastructures and the operations on them is the algebraic specification formalism ACT ONE [11]. We hardly need the data part for the definition of We-Met in LOTOS and will elaborate on the process part mainly.

A LOTOS specification consists of processes which communicate over gates. The entities communicated are events (also called actions) which are the atomic units of interaction and synchronization between processes. A process consists of behaviour expressions of which the simplest ones are events, internal actions and calls to other processes. Behaviour expressions can be composed by several operators. The most important ones are ";" and "$\gg$" for sequential composition and "[]" for choice. LOTOS has three operators for parallel composition: "||", "|[$g_1 ... g_n$]|" and "|||".

A	B	Effect
a !i	a ?j:Int	Processes A and B synchronize over action a and the value in variable i of process A is transmitted to variable j of process B.
a ?i:Int	a !j	The value in j is transmitted to i.
a !i	a !j	The synchronization over action a can only occur if the values in i and j match.
a ?i:Int	a ?j:Int	The variables i and j will after the synchronization contain an equal, but unspecified, value.

Figure 1: Different combinations of "!" and "?"

Processes which are composed by "||" run independently and do not synchronize at all. If two processes are composed by "|[$g_1 ... g_n$]|", they only synchronize over gates $g_1 ... g_n$. This means that if one of the processes wishes to perform action g_i, that it will have to wait for the other process to perform g_i also. Actions which are not in $g_1 ... g_n$ are performed independently of the other processes. Processes which are composed by "|||" synchronize on all actions.

An important aspect of the synchronization in LOTOS is the fact that it is non-directional and that it is not limited to two processes; any number of processes can participate in a single synchronization over an event. This allows for a *constraint oriented* specification style, which stands for a style where different processes put their own constraints on the possible sequences of events. If processes are composed, these constraints are composed also. This makes it possible to specify low-level components without knowledge of the environment in which they will be used.

If processes synchronize over an event they can also exchange data. There are two constructs for this exchange "*action !value*" to provide a value and "*action ?variable:type*" to receive a value. Figure 1 above gives an overview of the different combinations in which these construct can be used and explains the meaning of those combinations. Several values may be exchanged during a single synchronization point and these values may float in any direction.

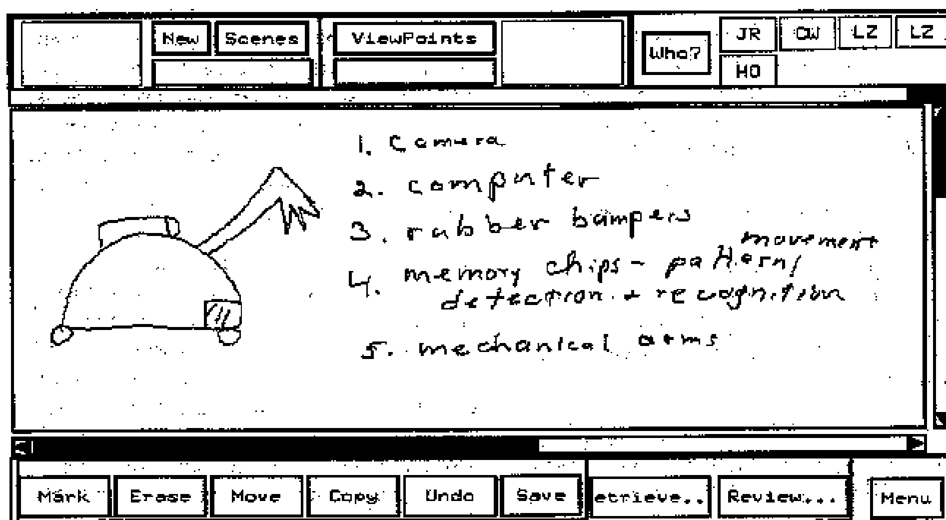

Figure 2: A We-Met screen

3 WE-MET

We-Met – which stands for *Window Environment – Meeting Enhancement Tools* – has been designed and implemented at IBM – Yorktown Heights and is a prototype pen-based tool designed to support both the communication and information retrieval needs of small group meetings [27].

We-Met provides a shared drawing area in which several users can work at the same time. When one person writes, the other participants see each stroke as it is completed. The system preserves temporal information, allowing users to travel backwards and forwards in time as they review a meeting. We-Met is a synchronous communication device, which means that the grain size of the actions is small and that participants see actions of others immediately without having to ask for it explicitly. It is We-Met's intention to support the communication process, not to structure it. For example, it does not impose floor control on the access to the drawing area or interface control. Figure 2 above contains a screen dump of a We-Met display.

Evaluating We-Met, we would say that it is a computerized version of a blackboard, with advantages and disadvantages over a real one. Its paper-like interface, its possibility to review a meeting and the ease of retreating on ones steps make it a very desirable communication device. The possibility to start a new scene or to return to an old one seems superfluous, as the time scroller can do it all, although the user interface of We-Met would then have to provide some tree-like scroller. Scrolling away for making private notes or to discuss something in a smaller group is a bad work-around for a real problem in the We-Met interface. We-Met needs elaboration on this point.

4 WE-MET IN LOTOS

In order to be able to describe We-Met in LOTOS we simplify its functionality to two commands: A user may draw a stroke or may go back in time.

4.1 The basic processes of We-Met

Figure 3 on the next page shows the intended layout of a We-Met system with two users. The idea is that a user process and its display synchronize fully on the action performed by the user, `ud_stroke` and `ud_back`. A *single* display and the central We-Met system synchronize on the display actions `dw_stroke` and `dw_back`. *All* displays synchronize on the action the We-Met system broadcasts, `stroke`. As We-Met is a communication device for synchronous communication, its implementation should enforce that all displays show the same information at all times. It is therefore that we request all displays to synchronize on broadcasted strokes, although this implies that the speed of the entire system is determined by the slowest display.

A `stroke` can both represent a stroke to add (to implement a drawing action) as one to delete (to implement scrolling back in time). Users may, for now, draw positive strokes only. We discuss extensions to our model of We-Met in Section 5, where we allow negative user strokes also.

4.1.1 The user and the display process

The processes `user` and `display` and their interaction are quite straightforward to describe in LOTOS.

```
process user [ud_stroke,ud_back] :noexit :=
    i; ud_stroke; user[ud_stroke,ud_back]
  []
```

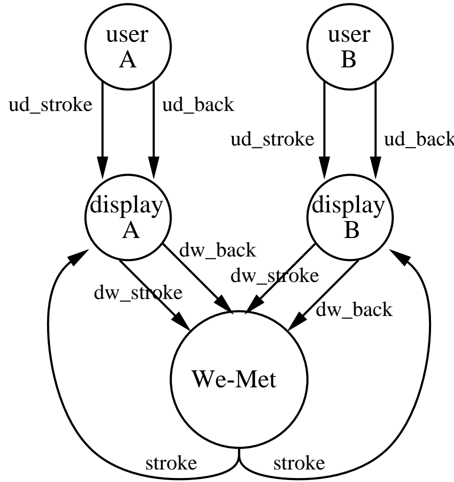

Figure 3: The layout of a We-Met system with two users

```
    i; ud_back;   user[ud_stroke,ud_back]
endproc
```

The `user` process can at each invocation perform the action `ud_stroke` or `ud_back`. In this choice, both actions are preceded by indistinguishable internal actions `i`. By doing so, we force that the user process alone determines which choice it makes. It can never be the case that some other process, with which the user process synchronizes over `ud_stroke` and `ud_back`, forces the actions to happen in a certain order. In other words, we specified that the user process alternates at its own pace between drawing a stroke and going back in time.

```
process display [ud_stroke,ud_back,
                 dw_stroke,dw_back,
                 stroke]
:noexit :=
    ud_stroke; dw_stroke;
    display[ud_stroke,ud_back,dw_stroke,dw_back,stroke]
  []
    ud_back; dw_back;
    display[ud_stroke,ud_back,dw_stroke,dw_back,stroke]
  []
    stroke; i;
    display[ud_stroke,ud_back,dw_stroke,dw_back,stroke]
endproc
```

The display simply passes user commands through to the central We-Met system, and reacts to strokes from We-Met with an internal action which represents the actual drawing or removing of a stroke on the physical display.

```
process one_display[dw_stroke,dw_back,stroke] :noexit :=
  hide ud_stroke, ud_back in
      display[ud_stroke,ud_back,
              dw_stroke,dw_back,
              stroke]
    |[ud_stroke,ud_back]|
      user[ud_stroke,ud_back]
endproc
```

Process `one_display` takes the processes `user` and `display` together and lets them communicate internally over their gates `ud_stroke` and `ud_back`. The whole can synchronize with the external world over the gates `dw_stroke`, `dw_back` and `stroke`.

4.1.2 The application process

The behaviour that has to be performed by We-Met as a reaction to the actions of the displays can also be specified as follows:

```
process application[dw_stroke,dw_back,stroke] :noexit :=
    dw_stroke; stroke;
    application[dw_stroke,dw_back,stroke]
  []
    dw_back; stroke;
    application[dw_stroke,dw_back,stroke]
  []
    stroke;
    application[dw_stroke,dw_back,stroke]
endproc
```

The process `application` reacts to the user action `dw_stroke` by echoing the stroke, it reacts to `dw_back` by echoing a negative stroke, and is willing to synchronize on `stroke`'s issued by eventual other application processes.

How this process `application` can be taken together with multiple `display` processes will be discussed in section 4.3. We first concentrate on an extension of the above with the actual strokes communicated.

4.2 The stack of strokes

The "back" command can be implemented by using a stack to store the strokes drawn. The back command then causes the last stroke to be popped from the stack and to be broadcasted as negative stroke.

This extension mainly concern the actions `ud_stroke`, `dw_stroke` and `stroke`, which have to be provided with ACT ONE terms that represent the strokes drawn or removed. The most interesting work on these strokes is performed by process `application`, which we present in more detail:

```
type Stroke_stack is Booleans, Naturals
  sorts STROKE, STACK
  opns  stroke: NAT, NAT, NAT, NAT, BOOL -> STROKE
        empty_stack: -> STACK
        push: STROKE, STACK -> STACK
endtype

process application[dw_stroke,dw_back,stroke]
                   (stack: STACK)
:noexit :=
    dw_stroke ?s:STROKE ;
    stroke !s ;
    application[dw_stroke,dw_back,stroke]
              (push(s,stack))
```

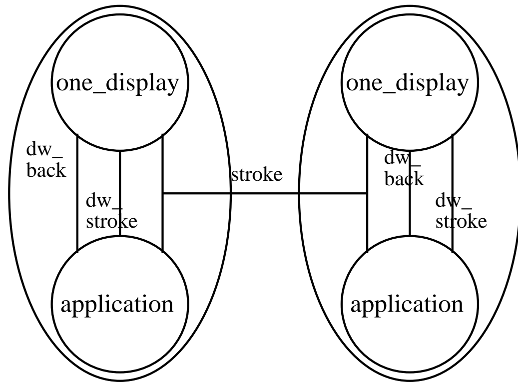

Figure 4: The localized solution

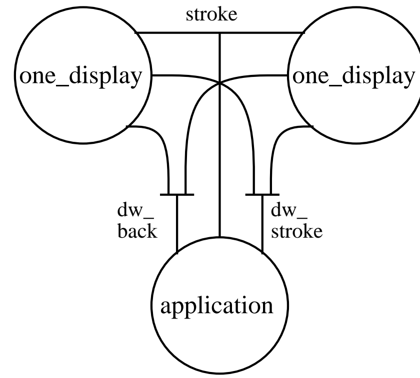

Figure 5: The centralized solution

```
  []
    dw_back;
    [stack = empty_stack] ->
      application[dw_stroke,dw_back,stroke]
                 (stack)
    [stack = push(stroke(n1,n2,n3,n4,true),stack')] ->
      stroke !stroke(n1,n2,n3,n4,false) ;
      application[dw_stroke,dw_back,stroke]
                 (stack')
  []
    stroke ?s:STROKE ;
    application[dw_stroke,dw_back,stroke]
               (stack)
endproc
```

The data types `STROKE` and `STACK` are declared and process `application` is modified to handle the data. Process `application` now receives strokes, stores them on its internal stack and redistributes them. If it receives a `dw_back` action, it broadcasts the stroke which was most recently pushed on the stack as a negative stroke. Note that we have modified the behaviour of the process for the case that the stack is empty: in that case the "back" command is just ignored.

4.3 The We-Met process itself

Now that we have defined the basic processes, we have to tie them together in the We-Met process itself. There are (at least) two possibilities for doing so which we will pursue in the sequel.

4.3.1 A localized solution

The synchronization between processes `one_display` and `application` is difficult to specify, due to the fact that `application` needs to synchronize with single displays on some actions and with all displays on other actions. We solve this by having one process `application` per display which handles the actions emitted by a display and which synchronizes with the other `application` processes over the broadcasted strokes. Figure 4 above depicts this architecture.

```
process We_Met [stroke] : noexit :=
    We_Met_unit[stroke]
  |[stroke]|
    We_Met_unit[stroke]
  |[stroke]|
    We_Met_unit[stroke]
endproc

process We_Met_unit[stroke] :noexit :=
  hide dw_stroke, dw_back in
      one_display[dw_stroke,dw_back,stroke]
    |[dw_stroke,dw_back,stroke]|
      application[dw_stroke,dw_back,stroke]
                 (empty_stack)
endproc
```

This solution does not have a central control, and we call it a localized solution. The semantics of the back command will in this solution be biased towards a solution where each user can remove strokes of his own only. If a user were to be allowed to remove strokes of other users also (as actually is the case in the We-Met system), each application process would have to store an own copy of the complete stroke history, and would have to keep this history up-to-date with the other stores.

4.3.2 A centralized solution

It is also possible to start just one process `application`, which handles the actions of all users. This solution is more biased towards the situation as it is in the real We-Met system, where each user can remove strokes of other users. Figure 5 above depicts the architecture intended, which we state in LOTOS as follows:

```
process We_Met [stroke] : noexit :=
  hide dw_stroke,dw_back in
    (   (   one_display[dw_stroke,dw_back,stroke]
          |[stroke]|
            one_display[dw_stroke,dw_back,stroke]
          |[stroke]|
            one_display[dw_stroke,dw_back,stroke] )
      |[dw_stroke,dw_back,stroke]|
        application[dw_stroke,dw_back,stroke]
                   (empty_stack) )
endproc
```

The processes one-display, which can all fire the actions dw_stroke, dw_back and stroke, are composed by |[stroke]| only. This means that they perform dw_stroke and dw_back independently of the others. These processes one-display are at an outer level composed with the application process with the operator |[dw_stroke, dw_back, stroke|]. This means that if any of the processes one-display wants to perform dw_stroke or dw_back, it needs to synchronize with process application. This is exactly as we wanted the synchronization to be.

Generalizing, in the localized specification strokes remain property of the user who issued them, where in the centralized version strokes become property of the group immediately.

In fact, the localized version of We-Met is formulated somewhat cumbersome, towards the derivation of the centralized version. The processes display and application can in the localized version be taken together without any loss of functionality. The display process becomes the application in that case.

Again, this centralized solution could also implement a semantics of the back command where a user can only remove strokes of its own. This would however induce a much more complicated data part. Strokes would need an identification of their issuer and the stack would have to be replaced by a linked list.

5 EXTENSIONS

We depart from the centralized version of We-Met (where users can discard strokes of anybody by going back in time) and discuss some possible extensions to it.

5.1 Deletion of selected strokes

Currently, users can only remove parts of the drawing by going back in time until some moment before that part was drawn. This also deletes anything drawn afterwards. If users should also be able to delete one or more selected strokes, we can model this by allowing the user process to send negative strokes as well.

Process application should then broadcast and stack these negative strokes, just like the ordinary strokes. The only modification necessary would be in the handling of the back command, which should send out a *negated* version of the last stroke on stack, as opposed to a *negative* stroke. This treats deletion of strokes just like addition: going back in time will reveal discarded strokes again.

5.2 Adding new users

Till now, we have assumed a fixed number of users. In a real system it should be possible for a user to enter a session after it started and to leave a session before it ends. We extend our specification with the possibility for users to ask for access. If they do, they receive the picture drawn till now, and are added to the session.

Most of the specification can be re-used to realize this extension of the functionality. The actions which can be performed by a user process are extended with start. The extended user process performs this action when it wants to join a We-Met session and next starts the ordinary user process. The actions of a display are extended with actions start, nw_display and comm_stack. The action nw_display is fired as soon as the display receives the action start of the user. The application process of the We-Met system will react on nw_display by communicating the stack to the display via the action comm_stack.

In order to obtain the possibility to add an at forehand unknown number of users, we introduce a process more_displays. This process is defined recursively and it starts a new process one_display_ext each time an action start is fired by a user process. In LOTOS[1]:

```
process more_displays[nw_display,comm_stack,
                      dw_stroke,dw_back,stroke]
:noexit :=
    one_display_ext[nw_display,comm_stack,
                    dw_stroke,dw_back,stroke]
  |[stroke]|
    more_displays[nw_display,comm_stack,
                  dw_stroke,dw_back,stroke]
endproc
```

5.3 Reviewing a meeting

Another important feature of the We-Met system is the possibility to review a meeting. We can add this functionality to the current specification of We-Met by introducing a separate archiver process.

This archiver has two phases, one to record a meeting and one to play it back again. While recording, it synchronizes on the strokes broadcasted by the We-Met process and stores these in a linear fashion. In the playback phase it accepts two commands, forward and backward, and uses the linear store of strokes as a vector with a pointer indicating where it currently is. On forward, the player emits the stroke under the pointer and restarts itself with an increased pointer. On backward, the player decreases the pointer, emits a negated version of the stroke pointed at, and restarts itself.

On having this functionality, a reviewer can scroll forward and backward through the entire meeting. We

[1] We use the same technique as demonstrated in the specification of the *daemon game* in [4, 8], but we are not entirely confident in it. The implementation of this process will in each case have to be lazy, as otherwise infinitely many processes more_display are started.

leave the actual LOTOS specification of this archiver to the imagination of the reader.

6 EVALUATION AND FUTURE WORK

The objective of this paper was to study whether specification formalisms for concurrent systems can also be used to specify the communicative aspects of CSCW systems in. Our first attempt to reach this objective was to develop a LOTOS specification of the CSCW tool We-Met.

We have indeed been able to describe different versions of We-Met in a clear and concise manner. These specifications describe the flow of information and the internal synchronization in an exact and high-level fashion. The synchronization primitives finally applied are however quite tricky; whether a casual reader of the specification understands all subtilities is questionable. It took us about two months to learn enough of LOTOS to be able to develop the final specification and to write this paper. Whether we judge LOTOS as an appropriate formalism to specify the behaviour of CSCW systems remains questionable, as LOTOS specifications turned out to be hard to develop and interpret.

We propose to continue this research by looking into more graphically oriented formalisms, like Graphical LOTOS (as used in [18]), SDL[1, 13], Estelle [7, 10] or Paradigm[14]. Specifications in these formalism will hopefully be easier to develop and interpret, but we will loose some of the possibilities to reason formally about specifications, what will make it harder to derive implementations from such specifications. We will also specify a CSCW tool that structures the communication and applies asynchronous communication in formalisms of both families.

After having performed this, we will be better qualified to judge the value of our initial idea's and will be better equiped to develop our own specification formalism, if necessary.

REFERENCES

[1] F. Belina and D. Hogrefe. Introduction to SDL. In *FORTE'88 proceedings*, September 1988.

[2] J.A. Bergstra and J-W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109–137, 1984.

[3] T. Bolognesi. The electric power of LOTOS – results of a joint academic/industrial experience. Presentation given at the *FORTE'92* conference, no paper, October 1992.

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

[5] R. Boumezbeur and L. Logripppo. Specification and validation of telephone systems in LOTOS. Technical Report TR-91-23, University of Ottawa, 1991.

[6] L. Brothers, V. Sembugamoorthy, and M Muller. ICICLE: Groupware for code inspection. In *CSCW'90 proceedings*, pages 169–181, October 1990.

[7] S. Budkowski and P. Dembinski. An introduction to Estelle: a specification language for distributed systems. *Computer Networks and ISDN systems*, 14, 1988.

[8] W.F. Chan and K. Turner. The daemon game in LOTOS. In *ESTELLE, LOTOS, SDL Draft Examples*. ISO, Turin, December 1986. ISO/TC97/SC21/WG1/FFDT – CCITT X/3.

[9] J. Conclin and M.L. Begeman. gIBIS: a hypertext tool for exploratory policy discussion. In *CSCW'88 proceedings*, pages 140–152, Portland, Oregon, September 1988.

[10] J-P. Courtiat. Estelle*: A powerfull dialect of Estelle for OSI protocol description. In S. Aggarwal and K. Sabani, editors, *Protocol Specification, Testing and Verification VIII*, pages 171–179. Elsevier Science Publishers, 1988.

[11] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1*, volume 6 of *EACTS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.

[12] P. Ernberg, T. Hovander, and F. Monfort. Specification and implementation of an ISDN telephone system using LOTOS. In *FORTE'92 proceedings*, pages 179–194, Lanion, France, October 1992.

[13] O. Færgemand and A. Olsen. Tutorial on new features in SDL-92. In *FORTE'92 tutorials*, pages 151–174, Lanion, France, October 1992.

[14] L.P.J. Groenewegen. Object-oriented cooperation control. Technical Report 91-03, Department of Computer Science, Leiden University, P.O. Box 9512, 2300 RA Leiden, the Netherlands, 1991.

[15] D Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[16] C.A.R. Hoare. *Communication Sequential Processes*. Prentice-Hall, 1985.

[17] ISO – Information Processing Systems – Open Systems Interconnection. LOTOS – a formal description technique based on the temporal ordering of

observational behaviour. Technical Report DIS 8807, ISO, 1987.

[18] H. Kremer, J. v.d. Lagemaat, A. Rennoch, and G. Scollo. Protocol design using LOTOS: A critical synthesis of a standardization experience. In *FORTE'92 proceedings*, pages 225–240, Lanion, France, October 1992.

[19] K-Y. Lai and T.W. Malone. Object-lens: A "spreadsheet" for cooperative work. In *CSCW'88 proceedings*, pages 115–124, Portland, Oregon, September 1988.

[20] M.D.P. Leland, R.S. Fish, and R.E. Kraut. Collaborative document production using Quilt. In *CSCW'88 proceedings*, pages 206–215, Portland, Oregon, September 1988.

[21] L. Logrippo, M. Faci, and M. Haj-Hussein. An introduction to LOTOS: Learning by examples. *Computer Networks and ISDN Systems*, 23(5):325–344, February 1992.

[22] T.W. Malone, K.R. Grant, F.A. Turbak, S.A. Brobst, and M.D. Cohen. Intelligent information sharing systems. *Communications of the ACM*, 30:390–402, 1987.

[23] R. Milner. *Communcation and Concurrency.* Prentice-Hall, 1989.

[24] S.A. Petri. Interpretations of net theory. Technical Report 75-07, Gesellschaft für Mathematic und Datenverarbeitung, 1976. Second, revised edition.

[25] W. Reisig. Petri nets in software engeneering. In W. Brauwer, W. Reisig, and G. Rozenberg, editors, *LNCS 255 – Petri Nets: Applications and Relationships to other Models of Concurrency*, pages 63–96. Springer-Verlag, 1986.

[26] M. Saeki, T. Kaneko, and M. Sakamoto. A method for software process modeling and description using LOTOS. In M. Dowson, editor, *First international conference of the Software Process – Manufacturing Comples Systems*, pages 90–104. IEEE computer society press, October 1991.

[27] C.G. Wolf, J.R. Rhyne, and L.K. Briggs. Communication and information retrieval with a pen-based meeting support tool. Technical Report RC 17842, IBM T.J.Watson Research Center, P.O.Box 704, Yorktown Heights, NY 10598, USA, 1992.