An Animation Facility to Simulate an Information and Communication System

Robert Croshere¹, Reind van de Riet² and Alco Blom²

¹ 4C International

² Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam

Abstract. Mokum is an object-oriented active knowledge base system with which the behaviour of a prototype of an information and communication system can be simulated. The system described in this paper is the animation facility which was added to Mokum. Through this facility it is possible to see what is happening with the simulation using several windows, the most interesting one being a window which acts as a kind of movie screen.

1 Introduction

An Information and Communication System (ICS) is a system comprised of people and autonomous components communicating with each other and manipulating information systems, such as databases and knowledge bases. We are interested in the process of designing and implementing such ICSs. In the recent past we have developed a language, CPL, for Conceptual Prototype Language, by means of which a specification of the ICS can be given in a way which is close to a specification in Natural Language.

Also we have developed an executable specification language: Mokum, (Manipulating Objects with Knowledge and Understanding in Mokum), which makes it possible to specify static and dynamic properties of objects sending each other messages, as in an ICS, and manipulating collections of objects. These objects can reason in a Prolog-like way. The Mokum system is implemented as a combination of Prolog and INGRES [vdR89].

It is hence very simple to define a prototype of an ICS in Mokum and study its behaviour by means of simulation. This paper deals with a new facility built into Mokum, namely an animation facility.

Using ideas from the field of Human Machine Interface (HMI) and Graphical User Interfaces (GUI), and the XPCE Prolog system [AW90], we have built this facility to optimally accomodate the designer of the ICS.

1. Objects are shown using coloured icons along with lists of attribute-value pairs, the icons are dependent of the type
2. Objects can be inspected and changed in a single menu-driven way
3. A message can be sent directly from the desk to an object and the effect can be seen immediately

4. Some objects can be put in a special window, upon which during simulation changes to these objects will continuously be shown. This facility is probably the most useful, because it gives the designer a real idea of the behaviour of the ICS
5. The invocation structure will be shown in a fourth window

The paper will give an introduction to Mokum by showing an example of a University with students, a student-administrator and a course-organizer, who all communicate and lead their lives. Then, the principles of GUIs will be described and what their relevancy is for the modelling of ICSs. The system built will be demonstrated using the University example. Finally, some implementation issues will be described, after which a conclusion will wrap up our main findings.

2 Mokum

Mokum, short for 'Manipulating Objects and Knowledge with Understanding in Mokum', was developed at the department of computer science at the Free University in Amsterdam. Mokum, described in [vdR89] and [WH90], is a knowledge base system. In Mokum, knowledge is represented in the form of objects and a conceptual model, both are stored in the INGRES database. This database, combined with the reasoning capabilities of Prolog, forms the knowledge base system. For small problems, with few objects, one can use a version of Mokum which operates without the INGRES database. This version is much faster because the connection Prolog – INGRES is a slow one. The graphical user interface, described in this paper, can be combined with both versions. In the sequel we will use the term Mokum standing for the version without INGRES.

2.1 The Mokum System

Currently, Mokum consists of three components: the Mokum Prolog kernel, as described in [GD89], the Mokum-To-Prolog translator, as described in [WH90] and the graphical user interface described in this paper.

In Mokum, objects represent entities in the UoD. Communication in the UoD can be modelled using messages. A program, written in the Mokum syntax, can be compiled, using the Mokum-To-Prolog translator, into Mokum Prolog kernel code. This code, which has the form of Prolog rules, can be loaded into a Prolog system containing the Mokum kernel, whereafter the user can manipulate the database and send messages from the Prolog command line. When loaded into XPCE Prolog, containing the Mokum kernel *and* the graphical user interface, the code can be animated on various aspects and the database can be efficiently manipulated through the interface.

2.2 Object-Oriented Concepts in Mokum

Most of the usual OO concepts are implemented in Mokum, one way or another. In this section we will explain their place in the Mokum system.

- Objects

An object in Mokum represents an entity of the UoD; it is internally represented in the form of a surrogate that, in principle, is not transferred to the user. In displaying objects it is used, however, for identification. Mokum does not provide a facility to designate certain attributes as key attributes.

- Attributes

In Mokum attributes and the range of these values are defined as follows: has_a 'attribute': 'range'. The range can be a primitive type (integer or string) and a user-defined type. The range can be restricted by a Prolog procedure, e.g. *birthdate-restriction* in Listing 1. Also, the value of an attribute can be computed by means of a Prolog procedure, e.g. *compute_age* in Listing 1. In addition the range can be a collection of objects of some type.

Attributes are defined just below the header in a type definition, restriction and derivation procedures can be defined in the procedure (proc) part of the type definition.

Listing 1

```

1  type person
2      has_a name: string           // primitive type as range
3      has_a address: addresstype  // user-defined type as range
4      has_a father: person        // user-defined type as range
5      has_a mother: person
6      has_a birthdate: int
7
8      // restricted attribute
9      restriction birthdate: birthdate_restriction
10
11     // computed attribute
12     has_a age: int := compute_age
13
14     has_a start_study_time: int
15     proc
16         compute_age:-
17             calculate_age(birthdate, Age), // system defined proc
18             age is Age.
19         birthdate_restriction:-
20             calculate_age(birthdate, Age),
21             0 <= Age, Age <= 100.
22     endproc
23     ...
24
25 type addresstype
26     has_a city: string

```

```

27     has_a street: string
28     has_a number: int.

```

• Class, Type

Objects are instances of types. In Mokum, a type defines the static and dynamic structure of a group of objects. All objects of a certain type are grouped together in a class; these classes are *not* visible to the outside world, but can be manipulated using the `new(...)`, `add.type(...)`, `destroy(...)` and `remove.type(...)` Mokum kernel calls (see following section). The value of a collection attribute is a subset of the class corresponding to the type of elements in the set. The object that contains a collection attribute is called the *keeper* of the collection; the collection keeper can also contain additional information about the collection in other attributes, like `number_of_elements` in the collection. In principle, the collection keeper has the duty to maintain integrity constraints for its collection.

In the header of a type declaration the user designates the supertype of the defined type, by stating `type 'subtype' is_a 'supertype'` (Listing 2). When no supertype is specified, i.e. `is_a ...` is omitted, the supertype is automatically assumed to be **thing** (which has no attributes or behaviour).

• Behaviour

The behaviour of an object is defined in the script part of the type declarations of the corresponding types of the object. The script definition consists of listing the states the object can be in for that type. A state an object is in is called an active state, an object can have a set of active states, (at least) one for each type the object has. The states are built up out of methods, called triggers in Mokum, that can be invoked in that state. There are two types of triggers in Mokum: triggers that are executed if a message is sent to the object with the trigger name in question, while the trigger is in the active state, and triggers that are invoked when a certain, designated, time occurs. The first are defined in the script with the `at.trigger 'name'` heading, the second with the `at.time 'time'` (see '`at.time time_of_promotion`' in Listing 2). This time is not real-time, but Mokum time i.e. simulation time, see next section. Triggers can optionally contain a check statement that checks if a certain predicate succeeds, if so, the message is accepted and the rest of the trigger is invoked. If the check fails, no actions are performed. When an object that contains two triggers in active states, having the same name, receives a message with the trigger name in question, there is no guaranteed order in which trigger will be tried first, thus it is best to keep trigger names unique.

When a type is added to an object, the first state in the script of the definition for that type becomes the active state for that type of the object. The `next(...)` kernel call allows the object to change its state; such a state change is called a transition.

Listing 2

```

1  type student is_a person
2      has_a student_number: int
3      has_a field: string
4      has_a average_grade: int
5      has_a year: int
6      has_a time_of_promotion: int          // time trigger
7      has_a do_a_course: int                // time trigger
8      has_a courses: collection_of course  // the courses to be done
9
10  script
11      state initial:                        // define state
12          at_trigger start_operations_of_student:
13              time(Now),
14
15              // determine the scheduled time for
16              // the trigger time_of_promotion
17              ProTime is 12 + Now,
18              ProTime to time_of_promotion,
19
20              // determine the scheduled time for the trigger do_a_course
21              ExamTime is 1 + Now,
22              ExamTime to do_a_course,
23
24              // change state
25              next(propaedeutical).
26
27  state propaedeutical:
28      at_time do_a_course:
29          (select(C in courses),           // pick a course
30              time(Now),
31              ExamTime is 2 + Now,
32              ExamTime to do_a_course,
33
34              // remove course from collection
35              [C] out_of courses,
36
37              new(M), add_type(M, message),
38              add_type(M, stud_co_message, [(course, C)]),
39
40              // send a message to the course organiser
41              send(course_organiser, enroll, M)
42          ; true),
43      next(propaedeutical).
44

```

```

45     at_time time_of_promotion:
46         new(M), add_type(M, message, []),
47         add_type(M, sa_co_st_message, []),
48
49         // send a message to the student administrator
50         send(student_administrator,
51             request_to_be_promoted_to_doctoral_stage, M),
52
53         // look at reply
54         Text from text of M,
55         (Text = 'you are promoted to doctoral stage',
56             next(doctoral) ;
57         Text = 'you are not promoted to doctoral stage',
58             next(propaedeutical)
59         ).
60
61     state doctoral:
62     ...
63
64     endscript.

```

3 Graphical User Interfaces

HCI, short for Human-Computer Interaction, has become an area of interest in computer science this last decade. The reason for the increased interest is three-fold: firstly, powerful workstations with bitmapped screens and pointing devices provide the new technology base; secondly, innovative applications have drawn attention to the computer's interactive, rather than its computational, abilities. Finally, the complexity of today's software has made better communication techniques a necessity, not a luxury. The study of HCI is a multidisciplinary field, various fields of computer science are involved as well as psychology (notably cognitive psychology and linguistics) fields. In this section we will concentrate mostly on computer science's contribution to HCI.

HCI consists mainly of user-interface design. In the following sections we will concentrate on graphical user interfaces, GUI's for short. It is acknowledged that designing a user interface is as much of an art as it is a science, although there are design principles that can help assure good human factors (*ergonomics*) in a design. In section 3.1 we will discuss these design principles, the rest of this section has the following structure: in section 3.2 we will discuss GUI's applied to databases, with examples, and in section 3.3 we briefly discuss program animation.

3.1 Design Principles

In [Eng67], the strong effect of different devices and techniques have been shown on speed of use and operator error rates. As more graphical user interfaces with similar functional capabilities become available, their success in the marketplace will be based increasingly on their ease of use. The ease of use, not the ease of implementation, is becoming the crucial design consideration. Ease of use isn't guaranteed by incorporating the following design principles [Fol82] in the interface design, although they are generally considered necessary for a successful design:

- Provide Feedback

Feedback is an essential ingredient in a conversation with a computer; without it, the user does not know how the computer is responding to the input (which can lead to frustration).

- Allow Users to Correct Mistakes

When the user makes a mistake he should be able to correct this mistake, that is, up until a certain degree, of course.

- Control Response Time

The response time that the user expects depends upon the action that the user issued to the machine; if this expected time is exceeded, the user can become frustrated.

- Structure the Display

A structured display can help the user understand and perceive the information displayed better. Structuring can be stimulated by various means: dividing the display into view areas and displaying similar information, each in its own area, makes it easier for the user to locate relevant information and provides an uncluttered and organised display. Discrimination of objects can be facilitated by color coding and representing objects as geometric shapes. Further, by only showing information that the user really needs, the user's efficiency, by perceiving the displayed information better, is increased.

- Minimize Memorisation

The need to memorise information during system use should be kept as low as possible, preferably totally avoided; the reasons for this are obvious.

- Aim for Consistency

When designing the interface, it is desirable to keep aspects like command language and display formats as uniform as possible. An example of consistency at the semantic level is making an operation, that is applicable to one class of objects, also applicable to another, similar, class of objects.

3.2 Interfaces to Database Systems

Because this paper is about an interface for a database system, we will limit ourselves to such systems.

Databases used to be accessible only through complex program-level interfaces, i.e. through detailed subroutines with long argument lists. Relational systems brought with them the possibility for end-user access through languages like SQL, but these were text-oriented and more of a high-level programming tool than an interface to a database. Still, most database systems don't utilise the graphics abilities that present computer systems provide. Fundamental questions that arise in building a database interface are:

1. How to access huge databases on a small computer screen?
2. How to graphically represent declarative structures, like constraints and queries?
3. How to graphically represent objects and relationships?

With the introduction of object-oriented databases that support complex objects, a fixed format for displaying the results of a query is no longer adequate. The desired format of the interface not only depends on the model that the underlying database uses, but also on the application domain of the database. Because a database package with interface is designed for many different application domains, the user should be allowed to customise the interface. Other desirable facilities of databases include the possibility to update objects on the screen and to browse through the database on different levels of data abstraction (for example schema and object level in OODB's).⁴

An example of a database GUI is ISIS. ISIS (Interface for a Semantic Information System), as described in [Gol86], is an interface for a semantic datamodel. In ISIS the user has the capability to design the schema, perform schema- and instance-level browsing and formulate queries. The user can specify a query by pointing to objects, attribute names and comparator operators, and others. The result of a query in ISIS is another subclass which can be inspected from the interface; one can zoom in on objects. ISIS maintains a uniform style of display throughout the system.

Another example is SIG (Smalltalk Interaction Generator). SIG, as described in [Mai84] is implemented in Smalltalk-80, but the design could be ported to other systems with window support, as well. The SIG system allows the user to customise the display of objects. SIG does not directly support schema design or querying, like ISIS. SIG does propagate updates in data to the screen, thus keeping information displayed on the screen up to date; ISIS does not provide this facility.

Related graphical interface systems include the Spatial Data Management System [Her80] and the Sembase semantic database system [Kin89].

3.3 Program Animation

As the human mind is strongly visually oriented, using pictures to propagate information to the user can be, depending on the information, much more ef-

fective than text. A person can discover complex relationships a lot faster via a picture than through text. Due to the increasing complexity and magnitude of modern software, a need for software tools to animate program operation has emerged. Program animation is used to explain and visualise the systems behaviour to the user, and can also serve as a debugging tool for the programmer. Since pictures are the best way to transport large amounts of related information, the behaviour of a system can best be illustrated by use of pictures. For the animation to be useful for someone who would like to inspect the program thoroughly, like a program debugger, facilities should be provided for: stopping the animation and continuing where the stop was issued, manipulating the speed of the animation and, most importantly, being able to modify parameters relevant for the animation during the animation session (i.e. through the interface). Various aspects of the program can be visualised, among the most used aspects are: control flow, data flow and data structures.

There are basically three different approaches to animating a program: animation through explicit programming, through low-level hooks and through an independent monitoring process. The first approach has to be done by the programmer himself, the other two can be done independently of the program, probably by a software tool.

- Explicit programming

The programmer inserts calls into the code that call a routine that updates the the display, these calls are inserted in places where interesting changes in the program state occur.

- Low-level hooks

Here, hooks are inserted into places/procedures where the state of the system is about to be changed. These hooks then propagate the changes to the display. The hooks are inserted into the low-level run-time software, so the program itself doesn't have to be modified.

- Monitoring

Here, an independent system monitors the interesting parts of the program; when the monitoring system picks up a change in the state of the system, it propagates this change to the display. As the previous approach, this approach also doesn't require modifying the program.

The system described in this paper animates programs written in the Mokum programming language. This animation consists of illustrating the data flow, which coincides with the control flow, and in a restricted manner, the program data structures. This is achieved by using hooks that catch interesting calls.

3.4 Further Readings

A comprehensive book on interactive graphics that also devotes attention to the human-computer interface is [Fol82]. In the IEEE workshops on Visual Lan-

guages articles on GUI's can be found; human-computer interaction is the main focus-point in the articles in SIGCHI proceedings and HCI congresses. In [Rae85] many aspects of graphical programming are discussed, and in [Fis89] the current state of HCI software is discussed. Working simulation and animation systems are discussed in [MM85] and [Kle88]. SIGGRAPH proceedings contain work in the field of computer graphics.

4 A GUI for Mokum

The animation system we built on top of Mokum, MGUI, was written in XPCE Prolog, and can be used for different purposes; it is mainly intended for two user groups:

1. Mokum programmers, to test and debug their programs, and
2. End-users, particularly information-analysts, who are interested in the information flow in the UoD, modelled by the Mokum program.

Currently, the emphasis is placed more on the first group, than on the second; this is due to the (still) limited animation capabilities present in MGUI.

Using MGUI consists of using three components that constitute the interface to the Mokum program:

- Database Window
- Control-Flow Window
- Animation Window

Globally, the user can manipulate the Mokum database which uses the conceptual model defined in the Mokum program, select objects out of this database, thus putting them on the animation window and monitor the behaviour of these objects via the animation and control-flow window during animation. The animation window thus functions as a link between the database window and the control-flow window. We will discuss these windows in the following sections.

4.1 The Example Program

MGUI uses a translated Mokum program as input. It is this program which defines the conceptual model; it is thus the key element around which the interface revolves. Throughout this section we will use one example Mokum program. It depicts a university UoD, consisting of students, a student-administrator and a course-organiser, and is, although limited, rich enough to display the features of MGUI. In Fig. 1 one can see all the types used. We have given definitions of person, address_type and student in Section 2. The other definitions are omitted for space reasons.

4.2 Sessions

During normal program use, at any moment two of the three major windows are visible. This is either the database and the animation window or the control-flow and the animation window. When the first option is chosen the user is able to access, manipulate and select objects in the database, this situation is then called a database session; likewise, the second option is called an animation session. Through commands in the menubar it is possible to switch between sessions. One can even suspend an animation session and start a database session, to later resume the animation session. When starting up, one automatically starts a database session. Figure 1 shows an example initial display with two windows; the left window called the database window and the right window called the animation window. Starting the example Mokum session a few objects are created automatically: VU : the university, Henk : the student administrator and Jan : the course organizer.

Fig. 1

4.3 The Database Window

The database window allows access to the database via the types in the type hierarchy, at class and instance level. Objects can be created, updated, and deleted from the Mokum database. The database window consists of two parts: a command area, and the display area containing the type hierarchy, i.e. a tree depicting the Mokum type hierarchy, as defined via the Mokum input program (Fig. 1). One can modify the objects in the database, but one cannot modify the type hierarchy, as it is part of the conceptual model defined in the Mokum input program.

After starting, we first create a person, by selecting the 'create' option in the popup menu of the person node in the hierarchy; this generates a create_instance window in which the attribute values are to be filled in. This is shown in Fig. 2.

Fig. 2

Some attributes are greyed out, these are referential attributes that can only be set via an update. A referential attribute refers to an object. The Start_Study_Time attribute is the (Mokum) time at which the person starts his study. As the simulated time is represented as an integer, "4" means : 4 units of time (one may think of days) after "now", determined by the Mokum CLOCK. After pressing 'OK', the object is created.

4.4 The Animation Window

In the previous section we created a few objects in the database. The objects of interest during the animation are put on the animation window. This can be done by selecting the 'select' option in the popup of the persons node on the database window, consequently selecting the attribute on which to sort the instances of type person, then selecting the desired instance in the Instance Browser and

finally selecting the option 'put in worksheet' from the popup menu for the selected instance or clicking the button 'put in worksheet' (Fig. 3). This results in an icon being put on the animation window corresponding to the person type, and labeled Alex (Fig. 4).

Fig. 3

We are interested to see how the person Alex becomes a student and how this affects the student administrator Henk. We also need the university object: VU. Therefore we put these three objects in the animation window and selected the menu item 'Show Attributes' from the popup menu attached to the objects Alex, Henk and VU. After moving the icons around a bit, we have the situation as in Fig. 4.

Fig. 4

Shown are the attribute-value pairs and the state which the object is in. Note also the ObjectNr: 'Value' field in the attribute windows. The ObjectNr is the object identifier or surrogate. In the current implementation it is an integer.

4.5 The Control-Flow Window

The control-flow window provides for a better understanding of the Mokum program (and of the modelled UoD), through animating certain aspects of the program. These aspects include: the flow of messages, which also happens to coincide with the control flow, the invoked triggers, the Mokum time and, through the animation window, the changing of type, and the updating of attribute values and state for selected objects. The control-flow window consists of three parts: a command area and a display area, like in the database and animation windows, and a button panel at the bottom of the display area (Fig. 5). In the following paragraphs these elements will be explained.

The Trigger Invocation Trees Initially, the display area contains only two nodes, labels 'USER' and 'CLOCK'. These nodes are roots of trigger invocation trees; a trigger invocation tree can be explained as follows. The nodes of the tree, except for the root nodes, are tuples, representing an invoked trigger of a certain object. These tuples have the form:

```
'object_number', 'type_of_object'
'trigger_name', 'mokum_time_of_invocation'
```

The `object_number` is a unique identifier for the object who's trigger is being invoked. This number enables the user to link objects in the control-flow window to objects in the animation window. The `type_of_object` is the type of the object that contains the invoked trigger, in its active state for that type, of course. The `trigger_name` is, as the name says, the name of the invoked trigger. The `mokum_time_of_invocation` is an integer stating the time of the Mokum internal clock, if existent, when the trigger was invoked. If no Mokum clock exists, i.e. no clock has been explicitly programmed by the user, the `mokum_time_of_invocation` field is always zero. These four elements thus represent an invoked trigger of an object; when this object, while executing this trigger, sends a message to another object and invokes one of its triggers, a new node is created, representing this new trigger, and a link is made by drawing a line between these two nodes, the new node being a child of the older one. In Mokum, when another trigger is invoked by a trigger that is executing, the executing trigger is suspended until the trigger it invoked finishes. A node can have more than one child nodes; after a child node finishes, the parent node resumes its trigger, if this invokes another trigger, another child node is created, etc. Because an object, that is executing a trigger for a certain type, can't have another trigger for that type invoked (to prevent deadlocks), the tree can never become a cyclic graph. To start a chain of trigger invocations, one must first

send an initial message. This message can come from two different sources: via a 'send' command on an object in the animation, or from the Mokum clock, sending a 'wake up' message to an 'at.time' trigger; this also explains the difference between the two trees. The tree with 'USER' as root node receives its initial messages from the user via a 'send' command, the tree with 'CLOCK' receives its initial messages from the Mokum clock. When a message is sent, but is not accepted, a node is added and deleted so quickly, that it is hardly perceived by the user.

When a trigger is being executed, the corresponding node in the invocation tree is highlighted; when the trigger is finished executing, it is unhighlighted. Triggers that are suspended, waiting for an invoked trigger to finish, still stay highlighted, because they are still active. There is always exactly one highlighted path in the tree, that is the path to the currently executing trigger. The root of the tree is always highlighted, of course. By using this highlighting scheme, where triggers that have finished executing remain on the screen, one can always see the history of trigger invocations.

4.6 Animating the example

The objects of interest have been selected on the animation window; we can now start an animation session by selecting the 'Ready' option in the menubar of the animation window. We want to get the system rolling by activating (changing their state from initial to something else) the three objects. First, we initialise the persons Alex and Henk. We do this by selecting the 'send' option from the popup menus attached to the icons, and invoke the trigger 'alive', which is shown in a dialog containing a list of active triggers. We also activate the university 'VU' by sending it the message with trigger 'start_operations'. We send these messages so to speak from the desk, to these objects.

At time 4, the at.time trigger for Alex will go off, because we have set the 'start_study_time' attribute to 4. When this trigger goes off Alex will then request to become a student by sending the student administrator a message (*wish.to.be.inscribed.as.student*). If the student administrator agrees, he sends Alex a message back with a confirmation that Alex is inscribed as a student (*start_operations_of_student*) which initialises Alex as a student. At this time Alex gets the type student. The Control-Flow Window now looks as in Fig. 5. One can also see the effects of Alex's behaviour on Henk's situation in Fig. 6 (the object VU has been removed from the window with the remove command for clarity).

Fig. 5

As one can see in the Control-Flow window, the trigger is busy executing the 'start_operations' and 'run_time' triggers of the VU university and the clock, respectively (because they are both still highlighted). It was therefore only possible to take this snapshot, by suspending the animation using the 'Stop' button. As one can see the 'Stop' button is greyed out. The first two unhighlighted children of the 'start_operations' node represent the executed triggers for initialisation for the course-organiser (start_operations_of_course_organiser) and the student administrator (start_operations_of_student administration), invoked in the 'start_operations' trigger.

In the Animation Window the icon and color of Alex can be seen changing (as can be clearly seen from figures 4 and 6). Also the attribute value items are expanded to show the attributes of type student. Furthermore, a new window

shows the collection of courses this student has to do to get his propaedeutical. An icon is put in the student collection of the student administrator Henk and also shown in the window named 'students' (Fig. 6).

Fig. 6

As time progresses, one can see the number of items in the courses collection window decrease as the student successfully passes the exams for the courses (Fig. 7).

Fig. 7

We have now shown in some detail the initial stage of the simulation. Other stages: how a student "requests to be promoted to the doctoral stage" and how that involves the help of Henk, the student administrator, and Jan, the course organizer, are not shown. It is easy to compute some statistics also and show the results by means of a few charts. This is common practice, however and not of enough interest to show here.

To conclude: evidently the animation tool is a useful one by means of which one can:

1. See what is happening
2. Control and influence the experiment from the desk
3. Create new objects and messages from the desk

4.7 Summary of the interaction

As was explained before, and clarified via the example, one is always in one of two MGUI sessions: either a database session or an animation session. One can switch between database and animation sessions using items in the animation and control-flow window command areas. The state of the MGUI system as a whole not only depends on the session the user is in, but also on the state of the Mokum database. This database can either be passive (i.e. no items in the database are active), active (i.e. some objects in the database are active) or suspended (which is the same as active, only the execution of triggers has temporarily been suspended). There isn't a 1-1 correspondence between sessions and

database states; one can, at different occasions, have different database states during a session. The state of the database can be manipulated via different media, notably through the 'Stop' and 'Continue' buttons in the control-flow window and through sending a message via the animation window. In Fig. 8, the relation between database states and MGUI sessions is given. Under the name of a session the list of possible database states during such a session is included between parenthesis. An arrow depicts a transfer from one state or session to another; between sessions the arrows are labelled with the command that causes the transfer; between database states the arrows are (possibly) connected via a dotted line to a session. The label on the dotted line depicts the the command/action issued to force the transfer, this command/action can only be issued from the session the dotted line is connected to.

Mokum Database State

MGUI Session

Fig. 8

4.8 Implementation

MGUI is programmed in XPCE [AW90]. XPCE is an object-oriented (OO) library, written in the C-language. It is designed to support the development of graphical user interfaces. XPCE assumes the presence of the X Window System and the programming language Prolog. The interface between XPCE and the windowing system is defined inside XPCE, there is a class for each type of object provided by the windowing system. The details of this interface are hidden from the application programmer.

XPCE contains a large collection of predefined classes and behaviour defined on these classes. Examples of predefined classes are: windows, dialogs, menus, user interaction objects (buttons, scrollbars), graphical objects, abstract data types, and so forth. Programmers can take advantage of this immediately and program a graphical user interface relatively fast as has been demonstrated in the current project. MGUI is programmed in about 40 pages of XPCE code. As explained in section 3.3, the mechanism used is by using hooks that catch certain Prolog calls.

4.9 Conclusion

In this paper we described an animation facility which gives a designer of an information and communication system (ICS) the possibility to see how his or her proposed system is functioning. Some general design criteria were developed and applied to an actual implementation of MGUI. That is the animated version of the Mokum system.

We have experimented with this system in a student environment in two ways: the example ICS was a system in which persons become students, in which students change state under the control of a student-administrator and a course-organiser. We also have let real students play with the system.

The implementation was done in such a way that we could profit very well from existing software in the form of the XPCE system.

In the future we want to couple Mokum to a formal language CPL, with which in a more formal way the static and dynamic properties of an ICS can be described. We also investigate how this can be combined with a Lexicon, derived from linguistics and with diagram-based tools to produce Mokum software [BvdR92]. The latter can then be run using the animation facility.

References

- [AW90] Anjewierden and Wielemaker. *PCE 3.7*. Amsterdam, 1990.
- [BvdR92] P. Buitelaar and R.P. van de Riet. The use of a lexicon to interpret er diagrams: A like project. In G. Pernal and A.M. Tjoa, editors, *Proceedings Entity Relationship Approach ER92*, pages 162–177. Springer, 1992.
- [Eng67] W. English. Display-selection techniques for text manipulation. *IEEE Transactions on Human Factors in Electronics*, 8(1), 1967.

- [Fis89] G. Fisher. Human-computer interaction software: lessons learned, challenges ahead. *IEEE Software*, January 1989.
- [Fol82] J.D. Foley. *Fundamentals of Interactive Computer Graphics*. Addison-Wesley, 1982.
- [GD89] M.V. Gamito-Dignum. Mokum (prolog) kernel users manual. Technical Report IR-196, Faculteit Wiskunde en Informatica, VU, Amsterdam, 1989.
- [Gol86] K. Goldman. Isis - interface for a semantic information system. In *Proceedings of the Third International Conference on the Management Of Data*, Washington D.C., 1986. ACM press.
- [Her80] C. Herot. Sdms: Spatial data management system. *ACM TODS*, 5(4), 1980.
- [Kin89] R. King. *Object-Oriented Concepts, Databases and Applications*. ACM Press, 1989.
- [Kle88] M Kleyn. Graphtrace - understanding object-oriented systems using concurrently animated views. In *Proceedings of the Third International Conference on Object-Oriented Programming Systems, Languages and Applications*, San Diego, September 1988.
- [MM85] B. Melamed and R. Morris. Visual simulation: The performance analysis workstation. *IEEE Computer*, August 1985.
- [MN84] D. Maier and P. Nordquist. Displaying database objects. *Expert Database Systems*, 1984.
- [Rae85] G. Raeder. A survey of current graphical programming techniques. *IEEE Computer*, August 1985.
- [vdR89] R.P. van de Riet. Mokum: an object-oriented knowledge base system. *Data and Knowledge Engineering*, 4, 1989.
- [WH90] A.F. Weesing and T. Hoogeveen. The mokum-to-prolog-kernel translator. Master's thesis, Faculteit Wiskunde en Informatica, VU, Amsterdam, 1990.