T U M

I N S T I T U T   F Ü R   I N F O R M A T I K

Formal Specification of CSCW Applications

Marjeta Frey-Pučko

TUM-I9745
November 97

T E C H N I S C H E   U N I V E R S I T Ä T   M Ü N C H E N

TUM-INFO-11-I9745-100/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Formal Specification of CSCW Applications

Marjeta Frey-Pučko

TUM-I9745
November 1997

Abstract

A formal method for specification of CSCW applications is suggested and its use for specification of the distributed multi-user editor Iris is shown in the report. We specify the interface layer of Iris, some elements of the access layer for its central part - structure editor - and some external components as an audio-video conferencing tool and an electronic mailer. We also show how a social protocol supported in a CSCW application can be considered in a formal specification and give methodological guidelines for the use of the method. Except the disadvantage that interfaces cannot be created dynamically, the proposed formal method based on the combination of CADT (Concurrent Abstract Data Types) and DistTL (Distributed Temporal Logic) is proved to be appropriate for specification of Iris.

Keywords: CSCW, Distributed Systems, Distributed Multi-User Editor, Formal Specification, Concurrent Abstract Data Types, Temporal Logic

Contents

Chapter 1

Introduction

1.1 Aims of the research

The concept of Computer-Supported Cooperative Work (CSCW) has been created to describe integration of computer communication and collaborative group work. In the last five years, a lot of research has been done in this field. The conducted research was mostly dedicated to multidisciplinary studies of group communication and collaboration, architectural models of CSCW systems and development of general groupware concepts. Not much work has been dedicated to formal specification of CSCW systems. Formal methods have generally been proved to be successfully used in different phases of software engineering, from requirements specification to automated testing. If applied, they could simplify also the development of CSCW applications and improve quality of their services.

CSCW applications can be constructed either from scratch or from existing components. Recently, the second approach is mostly used to provide flexible CSCW systems [Berry96, Dew96, Dou96]. However, a formal method for specification of CSCW applications should be usable for both approaches, since new components or some parts of applications even based on a few existing components are still created from scratch. Construction from exisiting components could be simplified if a unique formal description of available components would be available where only properties would be specified without details of implementation. Besides providing an unambigous description of components which in an informal way cannot be given, on this formal basis can be checked if the components can work together.

For formal specification of CSCW applications, we selected a combination of CADT (Concurrent Abstract Data Types) and DistTL (Distributed Temporal Logic). The advantage of this combination is the possibility to specify three different views of a distributed application:

- architectural view of interfaces for components and replications described by the architecture of the specification,
- the view of behaviour of the system and users described by temporal logic rules,

- the view of data described by abstract data types.

The usability and specification strength of new formal methods or use of existing ones are often tested by applying a method for specification of existing applications. This is also the aim of our research presented in this report. Such an approach has also been used for testing the suitability of LOTOS for specification of CSCW applications (Rekers and Sprinkhuizen-Kuyper [Rek96]).

1.2 Motivation for selecting Iris as an example

A testing use of a method can only give results which can be trusted if an example of a real CSCW application is complex enough and requires specification of different aspects. We selected the multi-user editor Iris as an example, since we want to specify the following aspects of user interaction and cooperation in a CSCW application:

1. user actions and the resulting external behaviour of the system,
2. combination of synchronous and asynchronous communication,
3. involvement of different communication media,
4. cooperation between users and cooperation awareness,
5. involvement of a social protocol,
6. provision of a final version of a document based on agreement of all involved users.

While the first three aspects are important in specification of a multi-user interface for any distributed multimedia application, the latter three are typical for CSCW applications.

In the next chapter, we explain reasons for adoption of the suggested approach. Sections 2.3 to 2.6 of chapter 2 describe theoretical background for our method and define the main constructs. The theoretical description is illustrated in chapter 3 by a practical specification example of a multi-user interface for a distributed editor. In chapter 4, we propose a methodology for the use of the method and discuss the possibilites to support the use by software tools and to improve its clarity. In the conclusion, we summarize our results and give some directions for future work.

Chapter 2

Specification method

2.1 Requirements

Considering the properties of CSCW environments and applications, the following requirements can be defined for a formal specification method which could appropriately describe these properties and be used in design and development of a CSCW application:

(1) Since the users of CSCW systems are often distributed in time and/or in space, multiuser systems in CSCW environments are naturally considered to be distributed systems (cf. [Elli91]). The method should therefore depict the distributed nature of the interface.

(2) For the same reason, it has to be able to express temporal relations and concurrency between actions.

(3) It has to be able to specify properties of reactive systems as for example liveness conditions, because most CSCW applications are reactive.

(4) To provide an abstract definition of structural elements of group activities [Dol91], types of cooperative actions should be differed in the interface specification (actions exchanging information between a user and the system, and actions supporting communication between users).

(5) Although we agree with Smith and Rodden [Smi93, Smi95] that access control in a CSCW system could not only be described by a pure social protocol, the specification method should also offer a minimal possibility to specify cooperation partly based on social rules.

(6) Since CSCW applications are usually multimedia applications, it should be able to specify different media involved.

(7) For construction of CSCW applications from components, it should provide compositionality.

(8) It should describe different views of an application: view of behaviour, view of architecture, and view of data.

(9) The possibility to describe only properties of the application independent of how it is executed should be given.

(10) Finally, the specification method should abstract the specified objects and their properties only to a level of abstraction which still allows clarity and conciseness.

The use of the method for specification of multiuser interfaces for CSCW environments implies some further requirements:

(11) The method has to be behavioural [Hart90], describing behaviour both of the users and the interface as they interact with each other, and should be able to express all user and system behaviour which is observable by an external observer.

(12) Further differentiation of actions should be provided: between user actions and system (computer) actions.

After the related research and motivation for the suggested approach, we present in the next section an algebraic formal specification method. Trying to meet the defined requirements, the method was developed by Frey and Pučko and proposed in [Frey97]. The version described in this report contains some minor improvements of the original method which introduce stronger differentiation of action types and parties executing actions. The actions may be of cooperative or noncooperative type and executed by a user or the system.

The method is based on enhancement and combination of two existing formal specification approaches: abstract data types and temporal logic. While an algebraic approach to description of CSCW applications and user interfaces is not new, the combination of these two approaches for specification of CSCW applications has not, to our knowledge, been described in the literature yet.

2.2 Related work and motivation

2.2.1 Formal methods for specification of CSCW applications

Several formal techniques, approaches and methods can be found in the literature which have been proposed for specification and development of applications for CSCW environments.

For specification of distributed *multimedia systems* in general, **Fischer** proposed the use of the standardized formal description technique Estelle with real-time extensions [Fisch96a, Fisch96b]. The semantics of the Real-Time Estelle was defined using a first-order temporal logic with time variables similar to Real-Time Temporal Logic (RTTL) of

Ostroff [Ost90]. **Kleuker et al.** suggested the specification language SL for specification and extensible design of multi-user multimedia systems [Kleu95, Kleu96a, Kleu96b]. The language is based on finite automata extended with variables. In addition, they offered a set of formal transformations which allow to extend an existing functionality of the system by means of adding new traces without entering deadlocks. They also discussed the relationship and transformation possibilities between the languages SL and SDL together with MSC [ITU-T93, ITU-T94].

Cooperation and *cooperative agents* were specified by **Diaz, Villemur and Vernadat** in VAL, a Petri net based description of dynamic agents [Diaz96]. The VAL specified behaviour was used to verify dynamic behaviour of a cooperative group in time according to entries and exits of users. Afterwards, the VAL description was translated into an Estelle description. Petri net extensions have also been sucessfully used for description of workflow management systems (**van de Aalst, van Hee and Houben** [Aalst94], **Agostini, de Michelis and Petruni** [Ago94]).

For specification of *asynchronous group communication*, **Benford and Palme** used an object oriented model to model groups exchaning information in an asynchronous way [Benf93]. *Synchronous group communication* was often specified in the standardized language LOTOS [ISO89]. **Kerbrat and Atallah** suggested in [Kerb95] an approach where architecture and components of a CSCW application are first described in COOP-SCAN and after that translated in LOTOS. Each component from the initial description is represented by a LOTOS process. Structured (with predefined communication paths among group members) or unstructured (without predefined paths) synchronous communication can be specified. **Rekers and Sprinkhuizen-Kuyper** [Rek96] combined LOTOS specification with specification by abstract data types in the way that the laws of operations for a particular data type are specified inside of LOTOS processes. The method does not structure communication.

A part of *Iris* functionality in combination with the functionality of the asynchronous activity support system TACTS was formally described by **Teege and Borghoff** [Teeg93] using finite state automata. The combined state automaton describes the main user interface for document objects.

The mentioned approaches mostly concentrate on specification of the executional view of CSCW applications. The view of properties is partly considered by Fischer [Fisch96a] in Real-Time Estelle semantics and by Rekers and Sprinkhuizen-Kuyper [Rek96] using abstract data types. Except the approach of Kleuker et al. [Kleu95], the approaches also have in common that they do not offer a precise methodology how a formal specification of a CSCW application can be constructed from an informal one. However, some of them are well supported by software tools, especially the approaches of Kerbrat and Atallah [Kerb95], and Diaz et al. [Diaz96].

2.2.2 Formal methods for specification of multi-user interfaces

Although several formal methods for description of user interfaces can be found in literature, they were mostly not created for specification of interfaces in CSCW environments.

A very interesting approach is the UAN technique offered by **Hartson, Siochi and Hix** [Hart90]. It is behavioural, easily usable for interface structure definition and able to express concurrency and temporal relations between actions, but it was created only for description of single-user interfaces. **Chi** [Chi85] presented four different axiomatic approaches to single-user interface specification which involve to some degree the notion of data abstractions. One of them can deal with concurrency in interaction. The method of **White** [Whit90] based on Petri nets can be considered as well suited for development of multi-user interfaces, particularly for specification of temporal relations. Its disadvantage is, however, its system-oriented character concentrating on time and use of system resources and not precisely specifying interaction between users and the system. **Johnson, Diaper and Hammond** [John91] extended a first order logic with temporal ordering to analyze concurrency and contention between multiple users of shared resources. Their method can generally be applied in groupware design, but it does not explicitly cover specification of CSCW aspects (types of actions, for example).

Some approaches are object-oriented. **Shen and Dewan** [Shen92] proposed a formal specification method based on sets of objects which can efficiently characterize collaborative properties of a system, Its purpose is mainly to specify the aspects of user interfaces related to access control. **Smith and Rodden** [Smi93, Smi95] also used sets of objects in a rather nonstrictly formal manner. **Bing and Li** [Bing97] defined a layered structure of a CSCW application. The upper three layers are interactive and consist of objects communicating with other objects in the same, higher or lower layer. Another formal approach was offered by **Graham and Urnes** [Grah92] where user interfaces were specified in the declarative language RVL by relations between program data structures and views on a display. The mentioned approaches also used several interesting CSCW concepts of interface design and coupling which we have adopted in the design of our specification method by merging and expressing them in a more formal way. The generalized editing model of collaboration where users interact with a collaborative application by concurrently editing its data structures we adopted from [Shen92], while the principle of collaboration aware components of a multi-user interface has already been used in [Smi93]. The idea of distributed implementation of interfaces which have to be previously specified separately from application programs was offered in [Grah92]. **Jacquot and Quesnot** offered in [Jac97] a detailed methodology for formal specification of user interfaces in the VDM language [Jon90], very similar to the Z language [Spiv89]. **Puerta** [Puer97] also used a formal approach consisting of a series of declarative models and supported by an interactive environment for user-interface development. An overview of tools for implementing user interfaces is given in [Dew93].

Models of user-system interaction can also be found in the literature. **Taylor** [Tayl88] represents user-system interaction as a hierarchically structured framework of communication loops, each operating according to its own interaction protocols. Some other models are based on production systems (**Bovair, Kieras and Polson** [Bov90]) or action grammars (as offered by, for example, **Reisner** [Reis81], or **Payne** [Payne84]). As an alternative to those two types of models, the PROCOPE semantic networks have been developed by **Poitrenaud** [Poit95].

2.2.3 Reasons for selection of CADT and DistTL

The main reason for selection of CADT and DistTL is that the selected formalisms if combined are able to specify three important views of a distributed application: the views of architecture, user and system behaviour, and data. The properties of the combination also meet all the requirements for a formal specification method defined in the first chapter. Particularly important is the ability of the method to describe only properties of an application without considering how it is executed. Among other algebraic methods for specification of properties, we consider CADT and temporal logic (TL) to be much more appropriate than CCS or LOTOS. The description of an interface with CADT and DistTL specifies structure and behaviour of a user interface distributed over a system. The structure of an interface is defined by a hierarchy of interface specifications and its behaviour is depicted by laws. Due to the mathematical basis, the temporal logic helps to eliminate problems of simultaneous access to resources and the absence of liveness. Despite that our method based on CADT and DistTL was developed for specification of applications with typical CSCW aspects, it has wider applicability for specification of distributed multi-user multimedia applications in general.

CADT differ from abstract modules, proposed in [Kroe87a] and extended in [Less95], in the definition of the temporal logic and in the fact that they do not contain any shared variables. Our distributed temporal logic is based on a partial order and describes properties of the partial order without construction of global states. In this way, it is different from linear temporal logic (LTL) [Kroe87b] which is the temporal logic of abstract modules, and other logics like TLA [Lam94] or concurrent temporal logics [Rei88]. Our temporal logic is similar to that described in [Knir95] except the definition of some logic operators. It offers logic operators to describe much more complex behaviour than the distributed temporal logics for specification of communicating sequential agents proposed by Lodaya, Ramanujam and Thiagarajan [Lod87, Lod92]. Our logic also has some similarities to the distributed μ-calculus [Huhn96].

2.3 Algebraic specifications

To specify the structure of a CSCW application, CADT contain three main concepts:

- an algebraic specification defining abstract data types,
- interface specifications describing properties of single (user) interfaces,
- compound interface specifications specifying how the interface specifications are grouped and working together.

Algebraic specification is a well-known specification method ([Ehr85, Wirs90]). In this section we do not describe this method in detail but only recall some fundamentals needed for introduction of concurrent abstract data types.

A signature is a pair $\Sigma = (S, OP)$ where S is a set of sort symbols, and OP is a set of operation symbols equipped with a mapping $sig : OP \to S^* \times S$. Each signature $\Sigma = (S, OP)$ defines a set of terms which are formed from free variables and operation symbols of Σ. Let X_s be a set of variables of sort s and $X = \bigcup_{s \in S} X_s$. The set of terms of sort $s \in S$, $T(\Sigma, X)_s$ is the least set containing:

- $x \in X_s$, and f where $f \in OP$ with $sig(f) = s$,
- $f(t_1, \ldots t_n)$ where $f \in OP$ with $sig(f) = (s_1, \ldots s_n, s)$, and $t_i \in T(\Sigma, X)_{s_i}$ for all $i = 1 \ldots n$.

An algebraic specification $SPEC = (S, OP, L)$ consists of a signature $\Sigma = (S, OP)$ and a set L of laws. A Σ-formula is

- a Σ-equation of the form $t = t'$ where $t, t' \in T(\Sigma, X)_s$ for some $s \in S$,
- a conditional Σ-equation $(l_1 \wedge \ldots \wedge l_n) \to l$ where $l_1, \ldots, l_n, l$ are Σ-equations,
- a definedness formula $\neg(l_1 \wedge \ldots \wedge l_n) \to \neg D_s(t)$ where $D_s(t)$ is a definedness predicate, $s \in S, t \in T(\Sigma, X)_s$ and $l_1, \ldots, l_n$ are Σ-equations.

A law is the universal closure $\forall x_1 : s_1 \ldots \forall x_n : s_n(l)$ of a Σ-formula l where $\{x_1 : s_1, \ldots, x_n : s_n\}$ are all free variables of l.

A Σ-algebra A consists of an S-sorted family of nonempty domains $(A_s)_{s \in S}$ and an operation $f_A : A_{s_1} \times \ldots \times A_{s_n} \to A_s$ for each $f \in OP$ with $sig(f) = (s_1, \ldots, s_n, s)$.

An assignment of Σ-variables X to a Σ-algebra A is a mapping $ass : X \to A$ with $ass(x) \in A_s$ iff $x \in X_s$. ass is canonically extended to $ass^* : T(\Sigma, X) \to A$, inductively defined by $ass^*(x) = ass(x)$ for $x \in X$ and $ass^*(f(t_1, \ldots, t_n)) = f_A(ass^*(t_1), \ldots, ass^*(t_n))$ for $f \in OP$.

For any Σ-algebra A, assignment ass, and Σ-formula l, the relation "*A satisfies l w.r.t. ass*" $(A, ass \models l)$ is defined as follows:

- $A, ass \models t = t'$ iff $ass^*(t) = ass^*(t')$.
- $A, ass \models D_s(t)$ iff $ass^*(t)$ is defined in A.
- $A, ass \models \neg l$ iff not $A, ass \models l$.
- $A, ass \models (l_1 \wedge l_2)$ iff $A, ass \models l_1$ and $A, ass \models l_2$.
- $A, ass \models (l_1 \to l_2)$ iff $A, ass \models l_1$ implies $A, ass \models l_2$.
- $A, ass \models \forall x : s(l')$ iff $A, ass_x \models l'$ for all assignments $ass_x : X \to A$ with $ass_x(y) = ass(y)$ for all $y \neq x$ and $ass_x(x) \in A_s$.

With each signature Σ and each set of laws L one may associate the class $Alg(\Sigma, L)$ of all Σ-algebras satisfying the laws L.

The initial semantics of an algebraic specification is a Σ-algebra $I \in Alg(\Sigma, L)$ where for all $A \in Alg(\Sigma, L)$ a unique Σ-homomorphism[1] from I to A exists (I is an initial algebra in $Alg(\Sigma, L)$).

In what follows we use the notation

sort s: $op_1, \ldots, op_n$;
based on $sp_1, \ldots, sp_m$;
opns
$op_1(v\_1_1 : s\_1_1, \ldots, v\_1_n : s\_1_n : pre\_cond\_1) : s\_1;$
$\vdots$
$op_k(v\_k_1 : s\_k_1, \ldots, v\_k_n : s\_k_n : pre\_cond\_k) : s\_k;$
laws $x_1 : s_1; \ldots ; x_p : s_p;$
$l_1; \ldots ; l_q;$
endsort s

to denote a specification (S, OP, L) where

- $S = (\{s\} \cup S_{sp_1} \cup \ldots \cup S_{sp_m})$,
- $OP = (\{op_1, \ldots, op_k\} \cup OP_{sp_1} \cup \ldots \cup OP_{sp_m})$,
- $L = (\{\forall x_1 : s_1 \ldots \forall x_p : s_p(l_i), i = 1 \ldots q\} \cup L_{sp_1} \cup \ldots \cup L_{sp_m})$.

2.4 Specification of interfaces

An interface specification defines the interface between a user and a CSCW application or between two communicating parts of the application. The user or another part of application can initiate actions of an interface and get responds from the application which can be initiated by the application or other users. An interface specification $IS = (SPEC, VAR, ACT, LAWS, INIT)$ consists of:

- an algebraic specification $SPEC = (S, OP, L)$,
- a set of variable symbols VAR equipped with a mapping $type : VAR \rightarrow S$ which assigns a type to each variable,
- a set of action symbols ACT together with a mapping $act\_sig : ACT \rightarrow S^*$ defining types of formal parameters and a mapping pre_cond which assigns to each action a a conjunction f_a of equations $t = t'$ where t, t' are terms of $T((S, OP), VAR \cup \{x_i : s_i,\ s_i \in act\_sig(a)\})$. The variables of $\{x_i : s_i,\ s_i \in act\_sig(a)\}$ are universally quantified in f_a,

[1] A Σ-homomorphism maps the domains and operations of one Σ-algebra to those of another Σ-algebra in such way that the structure of the algebra is preserved.

- a set of laws $LAWS$ which specify the behaviour of the interface. Laws are specified by a temporal logic introduced in section 2.6,
- a set $INIT$ of (S, OP)-equations $t = t'$ where $t, t' \in T((S, OP), VAR \cup X)$ and the variables of X are universally quantified. The equations of $INIT$ specify the initial state of the interface.

Actions of an interface specification may be initiated either by users or by users and the application. If the latter is true, the set of action symbols ACT is a result of the mapping $act\_type\_sig : (ACTb \times ACT\_PART) \rightarrow ACT$ where $ACTs$ denotes a basic set of action symbols without identifiers of the parties involved in the interaction, and ACT_PART denotes the set of party identifiers. It may, for example, be defined as { "u","s" } for action symbols "u:send" and "s:transmit", send, transmit $\in ACTb$, where the first action is initiated by the user and the second by the system. Since not only user but also system actions may change system states, the method integrates user and computer system concerns in the design of interactive systems as required by Blandford and Duke in [Bland97]. If a system action is not the result of a particular user action, it is assumed that it has been initiated by the system spontaneously.

Similarly, actions of different types can be differed by action type identifiers. The mapping $(ACTb \times ACT\_TYPE) \rightarrow ACT$ produces symbols of actions of cooperative and noncooperative type. For $send, transmit \in ACTb$ with $ACT\_TYPE = \{$ "c","n" $\}$ the action symbols "c:send" and "n:transmit" may be defined to specify a cooperative action of sending, for example, a message by a user to another user, and a noncooperative action of transmission of this message through the system.

An action in the specification can contain identifiers of both types, since the same action may describe interaction between a user and the system in the local interface and cooperation between users in the system-interface. This is a difference to an implementation, where, as Dewan states in [Dew96], an interaction event often triggers a collaboration event, and therefore two different events on different levels are needed. When both types of identifiers are required, the action symbol set is defined over the mapping $(ACTb \times ACT\_PART \times ACT\_TYPE) \rightarrow ACT$, $ACT = \{$ u:c:send, s:n:transmit $\}$, for example). The extensions of the action set definition may be used optionally depending on what kind of interaction is to be specified.

Variables of an interface specification denote the internal state of the interface. Laws specify how an interface reacts on initiations of actions and how the initiation of an action changes the internal state.

To distinguish different initiations of actions we introduce for each action symbol an initiation set: For each action symbol $a \in ACT$ exists a countable initiation set $In(a)$. Each a-initiation ai is equipped with an actual parameter list $par(ai) = (t_1, \ldots, t_n)$ where $t_i \in T((S, OP))_{s_i}$ for $i = 1 \ldots n$ and $act\_sig(a) = (s_1, \ldots, s_n)$. Each element of the initiation set corresponds to a possible initiation of an action.

The semantics of an interface specification $(SPEC, VAR, ACT, LAWS)$ is defined by an interface structure $(I, SEQU)$ where I is an initial algebra and $SEQU$ is a set of sequences of interface states. Informally, each sequence of $SEQU$ represents a possible execution of the interface specification. The behaviour of the interface specification is represented by its possible executions represented by sequences in $SEQU$.

An interface state $\eta = (\mu, R, \gamma, p)$ consists of:

- a variable valuation $\mu : VAR \rightarrow I$ which assigns to each variable $v \in VAR$ an element of the domain I_s where $type(v) = s$,
- a set R (ready set) containing initiations which are ready to execute ($R \subset \bigcup_{a \in ACT} In(a)$),
- an initiation γ of an action which is currently or next executed.
- a state predicate $p \in \{IN, PRE\}$ denoting whether η is a state during the execution of γ ($p = IN$) or the state before the execution of γ ($p = PRE$).

An interface structure $(I, SEQU)$ is a model of an interface specification iff I is an initial $SPEC$-algebra, $SEQU$ is the maximal set of sequences where all sequences $seq \in SEQU, seq = (\eta_0, \eta_1, \ldots)$ have the following properties:

(1) $\eta_0 = \{\mu_0, \emptyset, \gamma_0, PRE\}$ and $I, \eta_0 \models init^2$ for all $init \in INIT$.

(2) $\eta_{i+1} = \{\mu_{i+1}, R_i \cup \{ai\}, \gamma_i, p_i\}$ iff a user has started the initiation ai of action a and $\eta_i = \{\mu_i, R_i, \gamma_i, p_i\}$.

(3) $\eta_{i+1} = \{\mu_{i+1}, R_i - \{ai\}, ai, PRE\}$ iff $\eta_i = \{\mu_i, R_i, \gamma_i, IN\}$ is the last state of the execution of γ_i, $R_i \neq \emptyset$, $ai \in In(a)$, $I, \eta_i \models pre\_cond(a)$, and for all $ai' \in (R_i - \{ai\})$ it is valid that ai was initiated before ai'.

(4) $\eta_{i+1} = \{\mu_{i+1}, R_i, \gamma_i, p_i\}$ iff a message was sent or received and $\eta_i = \{\mu_i, R_i, \gamma_i, p_i\}$.

(5) for all $l \in LAWS$, and all $\eta \in sequ$ $I, \eta \models l$.

(1) characterizes the initial state of the interface specification. R contains no initiations and γ_0 is the first initiation of an action by users. (2) describes the situation, when users are initiating an action. (3) is the transition between states, when the execution of an initiated action starts. The actions of the ready set must be executed in the same order as they were initiated by the user. (4) describes the situation, when a message is sent or received. Finally, (5) states that laws must be satisfied. The changes of the variable valuation μ are defined by laws.

We use the notation

interface $name$: $ea_1, \ldots, ea_n, evar_1, \ldots, evar_v$;
based on $sp_1, \ldots, sp_m$;

[2] For any initial algebra I, state η, and temporal logic formula f the relation "*f is satisfied in η w.r.t. I*" ($I, \eta \models f$) is defined in section 2.6.

instances:

var_1:s_{var_1}; ...; var_p:s_{var_p};

actions:

$a_1(par\_1_1 : s\_1_1, \ldots, par\_1_n : s\_1_n : pre\_cond\_1);$

$\vdots$

$a_k(par\_k_1 : s\_k_1, \ldots, par\_k_n : s\_k_n : pre\_cond\_k);$

laws: vl_1:s_{vl_1}; ...; vl_q:s_{vl_q};

l_1; ...; l_r;

init: vi_1:s_{vi_1}; ...; vi_t:s_{vi_t};

i_1; ...; i_u;

endinterface *name*

to denote an interface specification $IS_{name} = (SPEC, VAR, ACT, LAWS, INIT)$ consisting of

- $SPEC = (S, OP, L)$ where $S = S_{sp_1} \cup \ldots \cup S_{sp_m}$, $OP = OP_{sp_1} \cup \ldots \cup OP_{sp_m}$, and $L = L_{sp_1} \cup \ldots \cup L_{sp_m}$,
- $VAR = \{var_1, \ldots, var_p\}$ and $type(var_i) = s_{var_i}$ for all $i = 1 \ldots p$,
- $ACT = \{a_1, \ldots, a_k\}$, $act\_sig(a_i) = (s\_i_1, \ldots, s\_i_n)$ and $pre\_cond(a_i) = \forall par\_i_1 : s\_i_1 \ldots \forall par\_i_n : s\_i_n(pre\_cond\_i)$ for all $i = 1 \ldots k$,
- $LAWS = \{\forall vl_1 : s_{vl_1} \ldots \forall vl_q : s_{vl_q}(l_1), \ldots, \forall vl_1 : s_{vl_1} \ldots \forall vl_q : s_{vl_q}(l_r)\}$,
- $INIT = \{\forall vi_1 : s_{vi_1} \ldots \forall vi_t : s_{vi_t}(i_1), \ldots, \forall vi_1 : s_{vi_1} \ldots \forall vi_t : s_{vi_t}(i_u)\}$.

The action names $ea_1, \ldots, ea_n$ and the variable names $evar_1, \ldots, evar_v$ are exported and may be used in laws of compound interface specifications containing the interface *name*.

2.5 Compound interface specifications

A compound interface specification $CIS = (SPEC, COMP, LAWS)$ consists of:

- an algebraic specification $SPEC$,
- a finite component set $COMP$ of interface and compound interface specifications,
- a set of laws $LAWS$ denoted in the temporal logic defined in section 2.6.

This way, a compound interface specification defines a hierarchy on interfaces. Compound interface specification at the top level of the hierarchy defines for each user or application part a compound interface or an interface. The user can initiate actions of this interface or the compound interface.

The semantics of a compound interface specification is defined by a compound interface structure (I, PO) where I is an initial algebra and PO is a set of partial order structures over interface states.

A partial order structure $po = (ST, F, N)$ consists of:

- an countable set of interface states ST,
- a relation $F \subseteq ST \times ST$ which is irreflexive, and the reflexive transitive closure of F defines a partial order on ST,
- a function N which assigns to each local state a name symbol of an interface specification to which it belongs.

Informally, the relation F is defined by sequential executions of interface structures and communications between interface structures. Figure 2.1 shows an example of a relation F. The compound interface structure consists of three interface structures and the F consists of a sequence of interface states for each interface structure. Communication between the interfaces induces dependencies between message transmission and reception states in F.

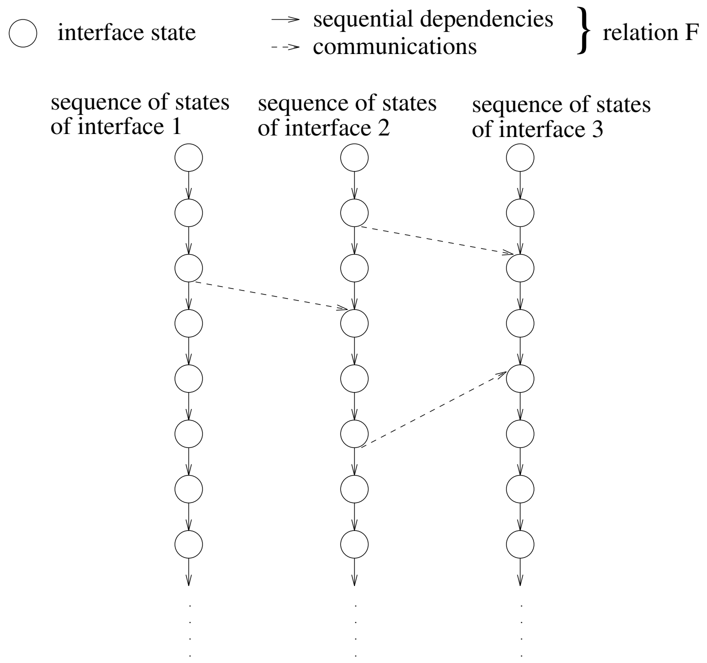

Figure 2.1: A partial order of three interfaces

The restriction of a partial order $po = (ST, F, N)$ to an interface or compound interface specification ci is a partial order $po|_{ci} = (ST_{ci}, F_{ci}, N)$ where $ST_{ci} = \{\eta \in ST,\ N(\eta)$ *is an interface specification name of* $ci\}$ and $F_{ci} = \{(\eta, \eta') \in F,\ \eta, \eta' \in ST_{ci}\}$. The restriction $PO|_{ci}$ contains for each $po \in PO$ the restriction $po|_{ci}$.

A compound interface structure (I, PO) is a model of a compound interface specification $(SPEC, COMP, LAWS)$ iff for all $comp \in COMP$ the following properties hold:

- $I|_{SPEC}$[3] is an initial $SPEC$-algebra,
- PO is a maximal set of partial orders where
  (a) all partial orders $po \in PO|_{comp}$ represent sequences of interface states, $(I|_{SPEC_{comp}}, \mathrm{PO}|_{comp})$ is an interface structure and a model of *comp* if *comp* is an interface specification,
  (b) $(I, PO|_{comp})$ is a model of *comp* if *comp* is a compound interface specification,
- For each $po \in PO$ each law $l \in LAWS$ *l is satisfied by po w.r.t. I* (see section 2.6).

The notation

interface *name*: $ea_1, \ldots, ea_n, evar_1, \ldots, evar_v$;
contains $comp_1, \ldots, comp_w$;
based on $sp_1, \ldots, sp_m$;
instances:
  $var_1 : s_{var_1}; \ldots; var_p : s_{var_p}$;
actions:
  $a_1(par\_1_1 : s\_1_1, \ldots, par\_1_n : s\_1_n : precond\_1)$;
  $\vdots$
  $a_k(par\_k_1 : s\_k_1, \ldots, par\_k_n : s\_k_n : precond\_k)$;
laws: $vl_1{:}s_{vl_1}; \ldots; vl_q{:}s_{vl_q}$;
  $l_1; \ldots; l_r$;
init: $vi_1{:}s_{vi_1}; \ldots; vi_t{:}s_{vi_t}$;
  $i_1; \ldots; i_u$;
endinterface *name*

denotes the compound interface specification $CIS_{name} = (SPEC, COMP, LAWS)$ containing

- an algebraic specification $SPEC = (S, OP, L)$ where $S = S_{sp_1} \cup \ldots \cup S_{sp_m}$, $OP = OP_{sp_1} \cup \ldots \cup OP_{sp_m}$, and $L = L_{sp_1} \cup \ldots \cup L_{sp_m}$,
- a set of components $COMP = \{comp_1, \ldots, comp_w, comp_{name}\}$ where $comp_{name} = IS_{name} = (SPEC, VAR, ACT, LAWS_{IS}, INIT)$. $SPEC$, VAR, ACT, and $INIT$ are defined as in interface specifications. $LAWS_{IS} = \{\forall vl_1 : s_{vl_1} \ldots vl_q : s_{vl_q}(l_i), l_i$ contains only symbols of $SPEC$, VAR, and ACT, $i = 1 \ldots r\}$,

[3] $I|_{SPEC}$ is the restriction of I w.r.t. $SPEC$. $I|_{SPEC}$ is the same as I except that only domains and operations occur which correspond to type and operation symbols of $SPEC$.

- a set of laws $LAWS = \{\forall vl_1 : s_{vl_1} \ldots \forall vl_q : s_{vl_q}(l_i), \forall vl_1 : s_{vl_1} \ldots \forall vl_q : s_{vl_q}(l_i) \notin LAWS_{IS}, i = 1 \ldots r\}$.

For the compound interface specification on the top of the hierarchy of interfaces we use a different notation:

system *name*:
consists of $comp_1, \ldots, comp_n$;
based on $sp_1, \ldots, sp_m$;
laws: v_1:s_1; ...; v_p:s_p;
  l_1; ...; l_q;
endsystem *name*

In contrary to the first notation a **system** specification does not contain a new interface specification. Furthermore, in the first notation each component of the **contains** expression belongs to the user of the compound interface. In the **system** specification each compound interface or interface specification of the **consists of** expression belongs to another user and laws specify how interfaces and, of course, users work together.

Each user interacts with one or more replications of this interface. We allow replication of interface and compound interface specifications in the **consists of** and **contains** expressions. Replicated interface specifications are denoted by $is[n...m]$ in an array-like notation. $is[n...m]$ specifies that there exist $m-n+1$ replications of the compound interface or interface specification is. The replications are denoted in laws by $is[n]$, $is[n+1]$, ..., $is[m]$. The use of a system component at the highest level is typical for cooperative applications when a replicated architecture is considered. Kerbrat and Atallah call it in [Kerb95] "collection component" and Dewan in [Dew96] "collaboration module".

2.6 Specification of interface behaviour

The laws of interface and compound interface specifications introduced in section 2.5 are depicted in a temporal logic, called *distributed temporal logic* (DistTL). This temporal logic allows to specify the behaviour of an interface and the cooperation between interfaces and, of course, users.

Linear time logics [Kroe87a] or branching time logics [Emer90] describe total orders (sequences) resp. trees. DistTL describes partial orders. In contrary to other logics describing partial orders (cf. [Rei88]) the partial order described by DistTL does not contain global states. It contains local states of interface structures.

Let $NAMES$ be the set containing all name symbols of specifications in $COMP$ of $CIS = (SPEC, COMP, LAWS)$. $IVAR_n$ are the variables and $IACT_n$ are the actions in CIS which are imported from specification $comp \in COMP$ with name n. Let $X = (X_s)$, $s \in S_{SPEC}$ be a finite set of variables, $XVAR_n = (XVAR_{n,s})$, $s \in S_{SPEC}$ and $XVAR_{n,s} = X_s \cup \{var \in IVAR_n, type(var) = s\}$.

There exist three different kinds of formulas in DistTL:

- interface state formulas (is-formulas)
- labeled is-formulas (lis-formulas)
- interface path formulas (ip-formulas)

An interface state formula of interface $n \in NAMES$ is inductively defined by:

- $pre.a(t_1,\ldots,t_m)$ and $post.a(t_1,\ldots,t_m)$ are is-formulas of n iff $a \in IACT_n$ with $sig\_act(a) = (s_1,\ldots,s_n)$ and $t_i \in T(\Sigma_{SPEC}, XVAR_n)_{s_i}$ for all $i = 1,\ldots,m$.
- $t = t'$ is an is-formula of n iff an $s \in S_{SPEC}$ exists such that $t, t' \in T(\Sigma_{SPEC}, XVAR_n)_s$.
- $\neg f$ and $(f \to f')$ are is-formulas of n iff f and f' are is-formulas of n.
- f is an is-formula of n iff f is an ip-formula of n.

A labeled interface state formula of interface specification n is of the form $n : f$ where $n \in NAMES$ and f is an is-formula of n.

An interface path formula of interface $n \in NAMES$ is inductively defined by: $\bigcirc_n f$, $\Box_n f$, and $(f\, before_n\, f')$ are ip-formulas of n iff f and f' are ip-formulas or lis-formulas.

A DistTL-formula of interface specification n is a universally closed lis-formula of n.

The semantics of DistTL is defined for any initial algebra I, any ass, any $po = (ST, F, N)$, any $\eta \in ST$ with $\eta = (\mu, R, \gamma, p)$ and $N(\eta) = n$, and each formula f of interface specification n by the relation "*f is satisfied by η w.r.t. I and ass*" ($I, ass, \eta \models f$). This relation is defined as follows:

- $I, ass, \eta \models pre.a(t_1,\ldots,t_m)$ iff $\gamma \in In(a)$, $par(\gamma) = t'_1 \ldots t'_m$, $vass^*(t_i) = ass^*(t'_i)$[4] for all $i = 1 \ldots m$, and $p = PRE$.
- $I, ass, \eta \models post.a(t_1,\ldots,t_m)$ iff $\gamma \in In(a)$, $par(\gamma) = t'_1 \ldots t'_m$, $vass^*(t_i) = ass^*(t'_i)$ for all $i = 1 \ldots m$, and an $\eta' \in ST$ exists with $(\eta, \eta') \in F$, $N(\eta') = n$, and $p' = PRE$, or $(\eta, \eta') \notin F^+$ for all $\eta' \in ST$.[5]
- $I, ass, \eta \models t = t'$ iff $vass^*(t) = vass^*(t')$.
- $I, ass, \eta \models \neg f$ iff not $I, ass, \eta \models f$.
- $I, ass, \eta \models (f \to f')$ iff $I, ass, \eta \models f$ implies $I, ass, \eta \models f'$.
- $I, ass, \eta \models n : f$ iff $I, ass, \eta \models f$.

[4] $vass^*$ is a mapping $vass^* : T(\Sigma_{SPEC}, XVAR_n) \to I$ with $vass^*(x) = ass(x)$ for $x \in X$, $vass^*(v) = \mu(v)$ for $v \in IVAR_n$, and $vass^*(f(t_1,\ldots,t_n)) = f_I(vass^*(t_1),\ldots,vass^*(t_n))$ for all $f \in OP_{spec}$.

[5] F^+ is the transitive closure of F.

- Let f be a formula of interface specification n':
  $I, ass, \eta \models \bigcirc_n f$ iff an $\eta' \in ST$ exists with $(\eta, \eta') \in F$, $N(\eta') = n'$, and $I, ass, \eta' \models f$.

- Let f be a formula of interface specification n':
  $I, ass, \eta \models \Box_n f$ iff $I, ass, \eta' \models f$ for all $\eta' \in ST$ with $(\eta, \eta') \in F^+$ and $N(\eta') = n'$.

- Let f be a formula of interface specification n' and f' be a formula of interface specification n'':
  $I, ass, \eta \models (f\, before_n\, f')$ iff for all $\eta'' \in ST$ with $(\eta, \eta'') \in F^+$, $N(\eta'') = n''$, and $I, ass, \eta'' \models f'$ an $\eta' \in ST$ exists with $(\eta, \eta') \in F^+$, $(\eta', \eta'') \in F^+$, $N(\eta') = n'$, and $I, ass, \eta' \models f$.

- $I, ass, \eta \models \forall x : s(f)$ iff $I, ass_x, \eta \models f$ for all assignments $ass_x : X \to I$ with $ass_x(y) = ass(y)$ for $y \neq x$ and $ass_x(x) \in I_s$.

A DistTL-formula f of n *is satisfied by* η *w.r.t.* I where $N(\eta) = n$ $(I, \eta \models f)$ iff $I, ass, \eta \models f$ for all assignments ass.

For any initial $SPEC$-algebra I and any partial order $po = (ST, F, N)$ a DistTL-formula f of n *is satisfied by* po *w.r.t.* I $(I, po \models f)$ iff for all $\eta \in po$ with $N(\eta) = n$ $I, \eta \models f$.

The laws of an interface specification $IS = (SPEC, VAR, ACT, LAWS, INIT)$ are formulas of DistTL which contain the name symbol of IS only. We can abbreviate the formulas of $LAWS$ by leaving out the name symbol.

Because an interface specification is interpreted by a set of sequences, its laws are satisfied by sequences and are formulas of linear time temporal logic [Kroe87a]. This way DistTL is an extension of a subset of linear time logic.

We will use some more abbreviations in the example of section 3.2:

- $a(t_1, \ldots, t_m)$ is equivalent to $pre.a(t_1, \ldots, t_m)$.
- $(f \wedge f')$ is equivalent to $\neg(f \to \neg f')$.
- $(f \vee f')$ is equivalent to $(\neg f \to f')$.
- $\Diamond_n f$ is equivalent to $\neg \Box_n \neg f$.
- $(f\, until_n\, f')$ is equivalent to $\neg(\neg f\, before_n\, f')$.
- $(f\, unless_n\, f')$ is equivalent to $(f\, until_n\, f') \vee \Box_n \neg f'$.
- $(f\, atnext_n\, f')$ is equivalent to $(\neg f' unless_n (f \wedge f'))$.
- If a compound interface specification contains a replicated interface specification IS, we may use quantification over variables which are of type *interface* IS in laws. This is equivalent to write down each law for each replication of IS. An example is given in specification of distr_editor.

Chapter 3

Case study: multi-user editor Iris

3.1 Functionality of Iris

Iris is a multi-user environment for group editing and collaborative writing of documents. It is described by Koch in [Koch95b] as an integrated environment with:

- *core applications* for concurrently editing documents, and
- different *external applications* for other collaboration and communication issues.

The most important core application is the Iris structure editor (istruct) for displaying and changing document structures represented by structure objects. Since objects may be of different media types (text, graphic, video), the environment includes several editor applications (GNU Emacs, a graphic editor and a combi-editor). For presentation of awareness information, there are several tools available: iuserinfo, ihostinfo and idocinfo (information on users, hosts and documents). Synchronous communication is supported by an external application for audio-video conferencing. For asynchronous direct communication, email and news are available.

The architecture of Iris is based on two layers: the *access layer* and the *user interface layer*. While the lower access layer is the core service for storing and accessing documents with high availability in wide area network environments (cf. [Koch95b]), the user interfaces in the upper layer provide user access to documents. The layers are separately implemented as processes. Documents are in the access layer replicated and their implementation is media dependent. Different sites involved in the document editing communicate over unreliable multicast.

The environment was developed at Lehrstuhl Angewandte Informatik - Verteilte Systeme, Institut für Informatik, Technische Universität München, and is still under improvement. A detailed description of Iris can be found in Iris-Online Manual [Iris96] or related publications, for example [Koch95a, Koch95b, Koch97].

To test the specification strength of our method, we decided to specify the central part of the environment consisting of istruct, the awareness tools and the text editor, and two external applications supporting communication: the email and the conferencing application. Our intention is to describe the interaction between a user of and the editing environment at the local single-user interface, and interaction (cooperation) between different users of the environment which can be observed on the level of the multi-user system-interface. The formal specification therefore mostly concerns the user interface layer of the Iris architecture. Since our method was developed to cover not only the architectural and the behaviour view of the interaction, but also the view of data, we also describe some elements of the access layer (storing documents or changing information in data structures of awareness tools, for example). In this regard, the Iris architecture cannot directly be mapped onto our architecture of interfaces.

3.2 Specification modules

3.2.1 Interface architecture

The first step in the use of our specification method is the definition of the interface architecture. The architecture depends on that which application calls and therefore "contains" another application.

¿From Iris structure editor all the applications may be called which support collaborative editing of a document (a text editor, email, audio-video conference and awareness tools). Therefore, the interface loc_istruct is the highest in the hierarchy of local interfaces and its specification has to contain specifications of hiearhically lower interfaces:

- loc_text_editor - an interface for a text editor,
- loc_email - an interface for an electronic mailer, which uses a hierarchically lower interface for a text editor loc_editor_in_email with the functionality adapted to composition of messages,
- loc_conf - an interface for an audio-video conferencing tool,
- loc_user_info, loc_host_info and loc_doc_info - interfaces for awareness tools.

As a result, it requires a compound interface in the following form:

interface loc_istruct:
contains loc_email, loc_text_editor, loc_conf, loc_user_info, loc_host_info, loc_doc_info;
based on *int*, *int(sequ)*, ... (other sorts related);
actions
  local actions for loc_istruct and actions for interaction with other applications
laws
  local laws for loc_struct and laws for interaction with other applications
endinterface loc_istruct;

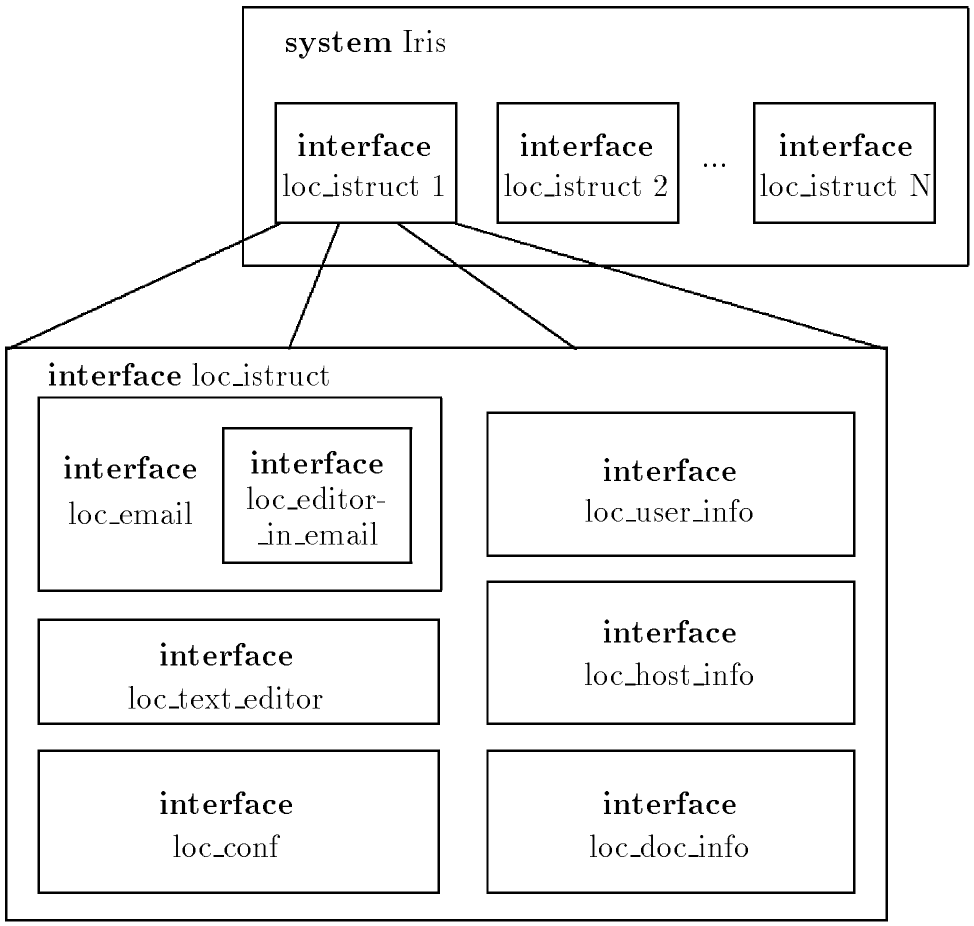

Figure 3.1: Architecture of interfaces

loc_struct containing the specified components is replicated for each of N users. The replicas loc_istruct1, loc_istruct2, ..., loc_istructN are on the highest architectural level composed into the system Iris (as shown in Figure 3.1):

A complete formal specification of loc_istruct cannot be given until all its parts are completely specified. In the next three sections we first define the sorts, interfaces and compound interfaces of email, text editor, audio-video conference and awareness tools. After that we give a detailed specification of the loc_istruct interface and, in the last section, an overview of the system-interface specification.

3.2.2 Specification of the email interface

email is a tool which supports asynchronous communication between users by messages. A message is explicitly sent by the sender and implicitly received by putting it into the mailbox of the recipient. An email interface typically consists of a sending and a receiving part. To send a message, an editor is needed where a new message can be written, and the information on the recipients and the subject of the new message has to be entered. To receive, a user needs a mailbox which contains messages in the order as they arrived.

A message consists of four parts:

(a) an identifier of the sender,

(b) identifier(s) of the recipient(s),

(c) a subject, and

(d) the text of the message.

A user who reads the contents of his/her mailbox may select a message considering the order of their arrival, and after that read or delete it. In the view to the mailbox contents, there is also information on senders and subjects available, taking into account the order of messages arrival again.

Specification of the described functionality requires two basic sorts: *message* which specifies operations on a message and *mbox* where operations on a mailbox are defined. First, we specify the sort *message*.

"empty_string" is a constant operation in the sort *string*, "empty_sequ_int" is a constant operation in the sort *sequ(int)* and "$\neq$" is an operation with signature (int):bool of sort *int*.

sort *message*: empty_message, insert_text, insert_subject, insert_recipients, get_text, get_subject, get_sender, get_recipients;
based on *string*, *int*, *bool*;
opns
  empty_message: *message*;
  is_without_sender(m: *message*): *bool*;
  is_without_recipient(m: *message*): *bool*;
  insert_text(m: *message*, s: *string*): *message*;
  insert_subject(m: *message*, s: *string*): *message*;
  insert_recipients(m: *message*, ids: *sequ(int)*): *message*;
  insert_sender(m: *message*, id: *sequ(int)*): *message*;
  get_text(m: *message*): *string*;
  get_subject(m: *message*): *string*;
  get_sender(m: *message*): *sequ(int)*;
  get_recipients(m: *message*): *sequ(int)*;
  convert_to_string(id: *sequ(int)*): *string*;
  make_message_string(m: *message*): *string*;
  is_substring(s: *string*, subs: *string*): *bool*;
laws s: *string*, id, ids: *sequ(int)*, m: *message*;
  get_text(empty_message) = empty_string;

get_subject(empty_message) = empty_string;
get_sender(empty_message) = empty_sequ_int;
get_recipients(empty_message) = empty_sequ_int;
get_text(insert_text(m,s)) = s;
get_text(insert_subject(m,s)) = get_text(m);
get_text(insert_recipients(m,ids)) = get_text(m);
get_text(insert_sender(m,id)) = get_text(m);
get_subject(insert_subject(m,s)) = s;
get_subject(insert_text(m,s)) = get_subject(m);
get_subject(insert_recipients(m,ids)) = get_subject(m);
get_subject(insert_sender(m,id)) = get_subject(m);
get_recipients(insert_recipients(m,ids)) = id;
get_recipients(insert_subject(m,s)) = get_recipients(m);
get_recipients(insert_text(m,s)) = get_recipients(m);
get_recipients(insert_sender(m,ids)) = get_recipients(m);
get_sender(insert_sender(m,id)) = id;
get_sender(insert_subject(m,s)) = get_sender(m);
get_sender(insert_text(m,s)) = get_sender(m);
get_sender(insert_recipients(m,ids)) = get_sender(m);
get_sender(m) = empty_sequ_int → is_without_sender(m);
get_sender(m) ≠ empty_sequ_int → ¬ is_without_sender(m);
get_recipients(m) = empty_sequ_int → is_without_recipient(m);
get_recipients(m) ≠ empty_sequ_int → ¬ is_without_recipient(m);
make_message_string(empty_message) = empty_string;
convert_to_string(empty_sequ_nat) = empty_string;
is_substring(s,empty_string);
is_substring(make_message_string(m), get_text(m));
is_substring(make_message_string(m), get_subject(m));
is_substring(make_message_string(m), convert_to_string(get_sender(m)));

endsort *message*;

The insert-operations insert a text, subject, sender and recipients of the new message. While "insert_sender" is usually performed automatically by the system, the other three operations are performed by a user. This is also true for "insert_subject", since we assume that no feature of automatic "reply" is available.

The is_without-operations are used for internal representation of "empty" recipients and senders. "get_text" and "get_subject" return a string with the text or subject of a message. The laws also define for which text the last subject, recipients and sender were inserted (similarly for other parts of a message).

"get_recipients" and "get_sender" return a corresponding identifier and may be used only in the system interface, because the index of the sending interface (i.e. identifier of the sender) and indices of the receiving interfaces (i.e. identifiers of the recipients) are in the hierarhically lower interfaces unknown.

The operation "make_message_string" produces a single string from all parts of a message. "convert_to_string" converts an identifier of the sort *sequ(int)* into a string. "is_substring(s,subs)" is true if subs is a substring of the string s.

Definition of the sort *mbox*:

"0" is a constant operation in *int*, "empty_sequ_string", "make_sequ" and "concat_sequ" are operations in *sequ(string)*. *sequ(string)* means an instantiation of the parameterized sort *sequ* over the sort *string* and returns a sequence of strings.

sort *mbox*: empty_mbox, num_of_messages_in_mbox, insert_message, delete_message, get_-message, get_subjects;
based on *int*, *message*, *string_sequ*, *bool*;
opns
empty_mbox: *mbox*;
num_of_messages_in_mbox(mb: *mbox*): *int*;
insert_message(m: *message*, mb: *mbox*): *mbox*;
delete_message(n: *int*, mb: *mbox*): *mbox*;
get_message(n: *int*, mb: *mbox*): *message*;
get_subjects(mb: *mbox*): *sequ(string)*;
laws mb: *mbox*, m: *message*, n: *int*;
num_of_messages_in_mbox(empty_mbox) = 0;
num_of_messages_in_mbox(mb) = n → num_of_messages_in_mbox(insert_message(m, mb)) = n + 1;
num_of_messages_in_mbox(insert_message(m,mb)) = n → get_message(n, insert_message(m,mb)) = m;
num_of_messages_in_mbox(mb) < n → get_message(n,mb)=empty_message;
num_of_messages_in_mbox(insert_message(mb,m)) > n → get_message(n, insert_message(m,mb)) = get_message(n,mb);
num_of_messages_in_mbox(insert_message(m,mb)) = n → delete_message(n, insert_-message(m,mb)) = mb;
num_of_messages_in_mbox(mb) < n → delete_message(n,mb) = mb;
num_of_messages_in_mbox(insert_message(m,mb)) > n → delete_message(n, insert_-message(m,mb)) = insert_message(m,delete_message(n,mb));
get_subjects(empty_mbox) = empty_sequ_string;
get_subjects(insert_message(m,mb)) = concat_sequ(get_subjects(mb), make_sequ(get_-subjects(m)));
endsort *mbox*;

The operation "num_of_messages" is defined to determine the number of messages in a mailbox. "insert_message" inserts a message into the mailbox. An outgoing message is assumed to be inserted in the mailbox only if the sender is included also in the list of recipients. "get_message(n,mb)" is used to access the message denoted inside the mailbox mb with the index n. "delete_message(n,mb)" deletes the message with the index n. n is defined in correspondence to the order of arrival assuming that in a mailbox with n elements the last element has the index n. If we want to delete from the mailbox a message with an index m greater than n, the missing k = m-n messages have to be inserted first, otherwise the first k messages in the mailbox would be deleted. The operation "get_subjects" returns subjects of all messages in the mailbox.

Since a process of editing is needed to enter the message text, a separate user interface has to be defined for a text editor. In the case of Iris and elm, the editor vi is used. However, the number of different vi operations available to a user could lead to enormous formal specifications which would describe all details of editing a text. Therefore we decided to specify only some basic operations needed to start and finish editing, and to save the results. These operations may have an abstract definition which also corresponds to the same operations in other text editors. We first define a sort *editor* and after that use it for definition of the interface loc_text_editor.

The sort *editor* describes the basic operations for editing. A user may call the editor and close it (operations "edit" and "close"). In addition, the user may save a file (the operation "save"). "open" is a system operation which loads a file which a user requires to edit. The operation "empty_ed" specifies an empty editor. The operation "changed" specifies if any changes have been made while editing a file. The operation "get_string_ed" returns the current contents of the editor.

Definition of the sort *editor*:

sort *editor*: open, close, edit, changed, save;
based on *file*, *string*, *bool*;
opns
  empty_ed: *editor*;
  open(f: *file*): *editor*;
  close(e: *editor*): *editor*;
  get_string_ed(e: *editor*): *string*;
  edit(e: *editor*, f: *file*): *editor*;
  changed(e: *editor*): *bool*;
  save (e: *editor*): *file*;
laws e: *editor*, f: *file*;
  get_string_ed(empty_ed) = empty_string;
  get_string_ed(open(f)) = get_string(f);
  ¬ changed(empty_ed);
  ¬ changed(open(f));
  get_string_ed(save(e)) = get_string_ed(e);
endsort *editor*;

In the definition of the interface loc_text_editor the instance act_file represents the file required by the user to be edited and the instance new_file represents a changed file after an execution of the action "save". The sort *file* is a sort defining some basic operations on files as we specified in [Frey97].

interface loc_text_editor: edit, close, open, save, new_file, act_file;
based on *editor*, *file*, *bool*;
instances
act_file: *file*;
new_file: *file*;
loc_ed: *editor*;
actions:
edit(); close(); open(f: *file*); save(f: *file*);
laws: f: *file*, e: *editor*;
/* *Postconditions of the actions* */
$post$.edit() $\rightarrow$ (loc_ed = open(act_file));
$post$.save() $\rightarrow$ (new_file = save(loc_ed));
/* *Edit only one file at the same time* */
edit() $\rightarrow$ ($\neg$(open() $\vee$ edit()) *until* close());
(*start* $\vee$ edit()) *before* (save() $\vee$ close());
close() $\rightarrow$ $\bigcirc$($\neg$(save() $\vee$ close()) *unless* edit());
/* *Safety* */
edit() $\rightarrow$ (save() *before* close());
init:
act_file = empty_file; new_file = empty_file;
endinterface loc_text_editor;

Not all operations generally specified for an editor by the interface loc_text_editor are actually needed for specification of the vi functionality, used inside of email. There are two basic differences: the currently edited file in email is always a new empty file, and the file need not to be saved explicitly by a user command save, since it is saved automatically by closing the editor. Therefore we define in the following another editor interface loc_editor_in_email which describes this simplified functionality:

interface loc_editor_in_email: edit, close, open, new_file;
based on *editor*, *file*, *bool*;
instances
new_file: *file*;
loc_ed: *editor*;
actions:
edit(); close(); open(f: *file*);
laws: f: *file*, e: *editor*;
/* *Postconditions of the actions* */
$post$.edit() $\rightarrow$ (loc_ed = open(new_file));

post.close() $\rightarrow$ (new_file = save(loc_ed));
/* *Edit only one file at the same time* */
edit() $\rightarrow$ ($\neg$ edit() *until* close());
(*start* $\vee$ edit()) *before* close();
close() $\rightarrow \bigcirc$ ($\neg$ close() *unless* edit());
init:
new_file = empty_file;
endinterface loc_editor_in_email;

Using the defined sorts and the interface loc_editor_in_email, the interface loc_email can be specified. The specification contains three instances:

- curr_sending_message denotes an instance of the message currently to be sent,
- new_message denotes an instance of a new message composed by a user, and
- mailbox is an instance of the user's mailbox.

s and s' represent strings of two different messages. rid and rid' represent sequences of recipient identifiers for different messages, and i and j identifiers of the sender and the recipient of the same message.

Actions available to a user are:

- "compose_message" performing all operations to compose a new message,
- "send_message" sending the composed message,
- "display_message" requiring from the system to show the contents of the selected message,
- "display_mbox_contents" requiring to show the contents of the user's mailbox,
- "cancel_message" discarding the message the latest composed, and,
- "del_message" deleting a message.

"DISPLAY_STRING" is a variable which specifies the actual presentation of a string (contents of the message or the mailbox) to a user required to be shown by the user actions "display_message" and "display_mbox_contents".

Since the following specification of the interface loc_email contains the interface loc_editor_in_email, it is a compound interface specification. It is divided into three parts: the specification of preconditions of actions, the specification of postconditions and the specification of interaction with the interface loc_editor_in_email.

We also specify the part of the system-interface referred to email. It describes the situation where the message m sent by the user[i] (if it is nonempty and equals to the instance curr_sending_message in loc_email) is at some unspecified future time put into the mailbox of the recipient user[j]. This asynchronous reception is described by the operator $\diamond$. curr_sending_message becomes after reception an empty message.

The operations "empty_sequ_int" and "is_element_in_sequ" are assumed to be defined in the sort *sequ(int)* and exported into the system-interface.

interface loc_email: compose_message, send_message, display_message, display_mbox_contents, cancel_message, del_message;
contains loc_editor_in_email;
based on *message*, *mbox*, *sequ(int)*, *bool*, *string*;
instances
curr_sending_message: *message*;
new_message: *message*;
mailbox: *mbox*;
actions:
compose_message(recipients: *sequ(int)*, subject: *string*); send_message(); display_message-(mn: *int*); display_mbox_contents(); cancel_message(); del_message();
laws: s,s': *string*, rid,rid',i,j: *int*, m: *message*, mb: *mbox*;
/* *Preconditions* */
$\neg$(compose_message(rid,s) $\wedge$ new_message) $\neq$ empty_message;
/* *Postconditions* */
(send_message() $\wedge$ m = new_message) $\rightarrow$ ((curr_sending_message = m $\wedge$ new_message = empty_message) *atnext* *post*.send_message();
post.display_message(i)$\rightarrow$ (DISPLAY_STRING (make_message_string(get_message(i, mailbox))) *until* (display_mbox_contents() $\vee$ display_message(j)));
post.display_mbox_contents()$\rightarrow$ (DISPLAY_STRING (get_subjects(mailbox)) *until* display_message(j));
post.cancel_message() $\rightarrow$ ((curr_sending_message = empty_message) $\wedge$ (new_message = empty_message));
(del_message(i) $\wedge$ mb = mailbox)$\rightarrow$ (mailbox = delete_message(i,mb) *atnext* *post*.del_message(i));
post.compose_message(s,rid)$\rightarrow$ ($\neg$ compose_message(s',rid') *until* send_message() $\vee$ cancel_message());
/* *Interaction with loc_editor_in_email* */
loc_editor_in_email: edit() $\rightarrow$ (loc_editor_in_email: close() $before_{loc\_editor\_in\_email}$ send_message());
compose_message(rid,s) $\rightarrow$ ($\diamond_{loc\_email}$ loc_editor_in_email:edit() $\rightarrow$ $\diamond_{loc\_editor\_in\_email}$ loc_editor_in_email: (*post*.close() $\wedge$ s' = get_contents(new_file))$\rightarrow$ $\bigcirc_{loc\_editor\_in\_email}$ new_message = insert_text(insert_subject(insert_recipients(empty_message, rid),s),s'));
init:
curr_sending_message = empty_message; mailbox = empty_mbox; new_message = empty_message;

endinterface loc_email;
/ The part of the system-interface related to email */*
system-interface: i,j: *sequ(int)*, m: *message*, mb: *mbox*;
loc_email[i]: (curr_sending_message $\neq$ empty_message $\wedge$ m = curr_sending_message $\wedge$ get_recipients(m) $\neq$ empty_sequ_int $\wedge$ is_element_in_sequ(j,get_recipient(m))) $\rightarrow$ ($\diamond$(loc_email[j]:mb = mailbox $\wedge \bigcirc_{interfaceuser[i]}$ loc_email[j]:mailbox = insert_message(insert_sender(m,i),mb)) $\wedge \bigcirc_{loc\_email[i]}$ loc_email[i]: curr_sending_message = empty_message);

Specifying the email interface we assumed that only one message may be edited at the same time. This is not true for some recent email tools which mostly allow to edit several messages whose number is usually not known in advance. Specification of this situation would require the possibility to describe dynamic creation of loc_editor_in_email interfaces that exceeds the current specification strength of our method.

3.2.3 Specification of the audio-video conference interface

The audio-video conference in Iris is supported by MBone tools *vat* (Visual Audio Tool) and *vic* (Video Conferencing Tool) [Koch97]. Each tool has its own interface which is integrated in the Iris audio-video conference compound interface. The session control is not a part of vic or vat and is built into the istruct and the system-interface.

Audio conference

An audio conference is a tool which supports the audio part of synchronous communication (conversation) between users distributed in space. The vat interface contains actions for user access to the microphone, for representation of the received audio signal to the user and for audio volume control. In our specification we assume the push-to-talk mode where the microphone always has to be activated before speaking and deactivated after speaking [Brau94].

The event in an audio conference which implies the basic specification element for the audio interface is a user speach. A user who wants to speak has to require activation of his microphone first. In our specification we call this user action "want_to_speak". If no other conference participant has the microphone activated at the moment and the initiator (chairman) of the conference agrees, the user's microphone is activated by the system action "s:activate_audio". After that the user may start to speak until he/she deactivates the microphone or the microphone is deactivated by the system for any reason. The user speach is therefore bounded with the actions "s:activate_audio" and "u:release_audio" (or "s:release_audio") and results in an audio message sent to all conference participants by multicast. To send and receive the audio message, the information on the identity of the speaker and identities of listeners is needed. We call an audio message extended with the additional information a protocol message and define a sort with the same name. The name has been selected in accordance to the ISO OSI terminology [ISO83] for description of communication primitives in distributed systems.

A protocol message in an audio conference consists of three parts:

- the contents (an audio message),
- an identifier of the speaker, and
- identifiers of the listeners.

The protocol message is actually constructed on the side of a recipient of an audio message, since all information which it contains has to be presented through an interface to a user.

In comparison to an email message, here the identifiers are only virtual parts of a message since the identifiers are of a different type and cannot directly be inserted into an audio message. The operation "create_audio_message" specifies the creation of an audio message. The operations "insert_speaker_id" and "insert_other_part_ids" executed by the system add the message the identity of the current speaker and the identities of other conference participants - its listeners. "get_audio_message" returns the contents of a protocol message , and "get_speaker_id" and "get_other_part_ids" return the identifiers of its author and listeners. The get-operations are actually internal operations of the system and are needed to access particular parts of a protocol message and to provide a more complete algebraic specification.

"empty_audio_message" is a constant operation in the sort *audio*, "0" in the sort *int* and "empty_sequ_int" in the sort *sequ(int)*.

sort *protocol_message*: empty_protocol_message, create_protocol_message, get_protocol-_message, get_user_id;
based on *audio*, *int*, *sequ(int)*;
opns
  empty_protocol_message: *protocol_message*;
  create_audio_message(pm: *protocol_message*, am: *audio*): *protocol_message*;
  insert_speaker_id(pm: *protocol_message*, id: *int*): *protocol_message*;
  insert_other_part_ids(pm: *protocol_message*, ids: *sequ(int)*): *protocol_message*;
  get_audio_message(pm: *protocol_message*): *audio*;
  get_speaker_id(pm: *protocol_message*): *int*;
  get_other_part_ids(pm: *protocol_message*): *sequ(int)*;
laws pm: *protocol_message*, am: *audio*, id: *int*, ids: *sequ(int)*;
  get_audio_message(empty_protocol_message) = empty_audio_message;
  get_speaker_id(empty_protocol_message) = 0;
  get_other_part_ids(empty_protocol_message) = empty_sequ_int;
  get_audio_message(create_audio_message(pm,am)) = am;
  get_audio_message(insert_speaker_id(pm,id)) = get_audio_message(pm);

get_audio_message(insert_other_part_ids(pm,ids)) = get_audio_message(pm);
get_speaker_id(create_audio_message(pm,am)) = get_speaker_id(pm);
get_speaker_id(insert_speaker_id(pm,id)) = id;
get_speaker_id(insert_other_part_id(pm,ids)) = get_speaker_id(pm);
get_other_part_ids(create_audio_message(pm,am)) = get_other_part_ids(pm);
get_other_part_ids(insert_speaker_id(pm,id)) = get_other_part_ids(pm);
get_other_part_id(insert_other_part_id(pm,ids)) = ids;
endsort *protocol_message*;

Although simpler, the specification is similar to the specification of the sort *message* used in the specification of loc_email. Specifying its external behaviour, we adopted a view to an audio-video conference as an exchange of protocol messages similarly to email. A protocol message is assumed to be completed and sent to other participants after "u:release_audio" or "s:release_audio". Adopting the same view, combination of static and temporal media can be described in the same formal specification in the most convenient way.

However, this does not exactly describe the continuous transmission/reception of an audio signal which should in a real audio conference always start immediately after "s:activate_audio". Namely, sound and video clips in temporal media have duration and, particularly in multimedia applications involving multiple users, are difficult to integrate with static media [Grah97]. The problem arises already in the specification phase. Using our specification method, a more precize description of transmission/reception could be achieved by one of the following approaches:

(a) Instead of a protocol message, the basic specification element could be a part of the message. The approach would require presentation of the message by an audio signal divided into a sequence of signal parts belonging to different time periods (similar to sampling periods of discrete-time signals [Oppen89]). The disadvantages are the large number of signal parts needed to model a real audio signal and the considerably lower level of abstraction than that used for specification of other aspects of interaction.

(b) The action of a message reception could be specified by a value of a logic variable which would describe reception of the message and its presentation to a user by having the value "true" in all interface states describing reception of the message and its presentation. The disadvantage of the approach is its implicitness since no explicite actions for reception and presentation of the message to a user are foreseen.

The approaches offer more natural representation of temporal media but, due to their disadvantages, they are of very limited practical use in examples of real applications. The most appropriate way to describe the temporal media could be the use of models of analog (continuous-time) signals which are completely out of scope of our work.

While the specification of the email interface includes only actions executed by a user, the specification of a conference interface requires also system actions. For example, in a video conference there are actually no user actions except that for changing window properties. System actions are needed to describe the behaviour of the video conference. We should therefore differ actions of both parties involved in the interaction by using the elements of the set $PART\_TYPE = \{$ "u","s" $\}$ of party identifiers as defined in section 2.4. To the audio part of the conference interface specified in the next subsection, the following user and local system actions are related:

- As already mentioned, "u:want_to_speak" is a user action which requires activation of the microphone from the system. The protocol primitive related to this action is in the literature often called "get_ floor" (as, for example, in [Kerb95]). "s:activate_audio" is a system action of activation of the microphone. "u:release_audio(pm,am)" and "s:release_audio(pm,am)" are actions of deactivation of the microphone. The actions have two parameters: am is an audio message of the sort *audio* and g is a video of the sort *graphic* which are generated as a result of the release-actions. The audio message contains the complete user speach and the video presents its behaviour while speaking. "s:release_audio(am,g)" is executed in the situation where the user, despite having the microphone activated, for any reason does not start to talk in considerable time or his/her speach has to be interrupted because of unacceptable social behaviour. In those cases, the microphone has to be deactived from the side of the system by "u:release_audio(am,g)".

- The system action "s:reproduce_audio(pm,am)" presents all parts of a protocol message to a user. While the identifiers of the speaker and other conference participants are shown on the screen, the audio message of the current volume is transformed into a form which can be received (i.e. heared) by a user.

- "u:set_volume(am)" is a user action for changing the volume of the audio.

Video conference

The basic sorts for specification of the video part of the conference interface are *window* and *window_sequ*.

A window presenting a conference participant consists of two parts: a name (identifier) of the participant and a video showing the participant during the conference. The sort *window* defines operations which a system executes to provide names and videos of participating users actually shown in the interface loc_conf. The operations "insert_user_name" and "insert_user_video" insert the name and the video of the user currently presented by the window. "get_user_name" and "get_user_video" return the inserted parts of the window.

The operation "0" in the following definition of the sort *window* is a constant operation in the sort *int* and the operation "empty_picture" is an operation in the sort *graphic*.

sort *window*: empty_window, insert_user_name, insert_user_video, get_user_name, get_user_picture;
based on *graphic*, *int*;
opns
  empty_window: *window*;
  insert_user_name(w: *window*, name: *int*): *window*;
  insert_user_video(w: *window*, g: *graphic*, name: *int*): *window*;
  get_user_name(w: *window*): *int*;
  get_user_video(w: *window*): *graphic*;
laws w: *window*, g: *graphic*, name: *int*;
  get_user_name(empty_window) = 0;
  get_user_video(empty_window) = empty_picture;
  get_user_name(insert_user_name(w,name)) = name;
  get_user_name(insert_user_video(w,name)) = get_user_name(w);
  get_user_video(insert_user_video(w,g,name)) = g;
  get_user_video(insert_user_name(w,name)) = get_user_video(w);
endsort *window*;

Windows containing videos of conference participants and their names are opened in that order as the participants join the discussion for the first time. In an abstract view, the opening procedure creates a sequence of windows. Here, the windows are assigned numbers (elements of the sort *int*) specifying the order as the windows have been added to the sequence.

A sequence of windows is described by the sort *window_sequ*. The operations "add_window" and "remove_window" add to and remove windows from a sequence. "num_of_windows" returns the current number of windows in a sequence. "get_window" returns the contents of a window in the sequence.

Operation "0" is a constant operation in the sort *int*. As already defined, "empty_window" is an operation in *window*.

sort *window_sequ*: empty_window_sequ, num_of_windows, add_window, remove_window, get_window;
based on *window*, *int*;
opns
  empty_window_sequ: *window_sequ*;
  num_of_windows(ws: *window_sequ*): *int*;
  add_window(w: *window*, ws: *window_sequ*): *window_sequ*;
  remove_window(wn: *int*, ws: *window_sequ*): *window_sequ*;
  get_window(wn: *int*, ws: *window_sequ*): *int*;
laws ws: *window_sequ*, w: *window*, wn: *int*;
  num_of_windows(empty_window_sequ) = 0;

num_of_windows(ws) = n → num_of_windows(add_window(w,ws)) = n + 1;
num_of_windows(add_window(w,ws)) = n → get_window(n,add_window(w,ws)) = w;
num_of_windows(ws) < n → get_window(n,ws) = empty_window;
num_of_windows(add_window(w,ws)) > n → get_window(n,add_window(w,ws)) = get_-window(n,ws);
num_of_windows(add_window(w,ws)) = n → remove_window(n,add_window(w,ws)) = ws;
num_of_windows(ws) < n → remove_window(n,ws) = ws;
num_of_windows(add_window(w,ws)) > n → remove_window(n,add_window(w,ws)) = add_window(w,remove_window(n,ws));

endsort *window_sequ*;

The following user and local system actions are related to the video part of the interface loc_conf:

- By the system action "s:show_windows(ws,g)" a sequence of windows is shown to a user. The first parameter represents the sequence of the sort *window_sequ* and the second the current window size of the sort *graphic*. Windows are configured to "follow the speaker". For the currently speaking user, a video is shown in a window [MBo96]. Here, we assume that the number of conference participants is small enough to display windows for all participants.
- "s:close_windows" is a system action which closes a sequence of windows after that an invited user left the conference or the conference has been closed by the initiator.
- "u:redefine_window(g)" is a user action which changes the window properties, for example the window size denoted by the parameter g of the sort *graphic*.

The action "s:open_conf" opens a local conference upon an initiation from loc_istruct. "u:close_conf" and "s:close_conf" close the conference depending on that if the conference was closed localy by one of the invited users or globally by the initiator.

The operation "get_volume" returns the current volume and is assumed to be defined in the sort *audio*. "get_size" from the sort *graphic* returns the current size of a picture in a window. All operations taking care of details in transmission and processing of audio and video signals (as described, for example, in [Weg95] or [Will97]) are assumed to be hidden in the sorts *audio* and *graphic*.

The interface loc_conf is specified as follows:

interface loc_conf: s:open_conf, u:want_to_speak, s:activate_audio, u:release_audio, s:release_audio, active_audio, s:reproduce_audio, active_video, s:show_windows, windows, u:close_conf, s:close_conf;
based on *audio*, *graphic*, *protocol_message*, *window*, *window_sequ*, *bool*;
instances:

curr_volume: *audio*;
curr_picture_size: *graphic*;
windows: *window_sequ*;
active_conf, active_audio, active_video: *bool*;

actions: s:open_conf(); u:want_to_speak(); s:activate_audio(); u:release_audio(am: *audio*,g: *graphic*); s:release_audio(am: *audio*, g: *graphic*); s:reproduce_audio(pm: *protocol_message*, am: *audio*); u:set_volume(); u:redefine_window(); s:show_windows(ws: *window_sequ*, g: *graphic*); u:close_conf(), s:close_conf();

laws: am, am': *audio*, g, g': *graphic*, pm: *protocol_message*, ws: *window_sequ*;

/* *Sending part:* */
/* *Preconditions* */
pre.s:open_conf() $\rightarrow \neg$ active_conf;
(*pre*.s:close_conf() $\vee$ s:close_conf()) $\rightarrow$ active_conf;
pre.s:activate_audio() $\rightarrow \neg$ active_audio $\wedge \neg$ active_video;
(u:release_audio() $\vee$ s:release_audio()) $\rightarrow \neg$ active_audio $\wedge \neg$ active_video;
active_audio $\rightarrow$ active_conf;
active_video $\rightarrow$ active_conf;
/* *Postconditions* */
post.s:activate_audio() $\rightarrow$ (active_audio $\wedge$ active_video) *until* (*pre*.u:release_audio() $\vee$ *pre*.s:release_audio());
(*start* $\vee$ *post*.u:close_conf() $\vee$ *post*.u:release_audio() $\vee$ *post*.s:release_audio()) $\rightarrow$ ($\neg$ active_audio $\wedge \neg$ active_video) *unless* *pre*.s:activate_audio();
(*start* $\vee$ *post*.u:close_conf() $\vee$ *post*.s:close_conf()) $\rightarrow \neg$ active_conf *unless* *pre*.open_conf();
/* *Audio and video* */
u:want_to_speak() $\rightarrow \diamond$ s:activate_audio();
s:activate_audio() $\rightarrow \neg$ u:want_to_speak() *until* (u:release_audio() $\vee$ (s:release_audio());
/* *Only one conference active at the same time* */
s:open_conf() $\rightarrow \neg$ s:open_conf() *until* (u:close_conf() $\vee$ s:close_conf());
start $\vee$ u:close_conf() $\vee$ s:close_conf() $\rightarrow$ s:open_conf() *before* (u:close_conf() $\vee$ s:close_conf() $\vee$ u:want_to_speak() $\vee$ s:activate_audio() $\vee$ u:release_audio() $\vee$ s:release_audio());

/* *Receiving part:* */
/* *Preconditions* */
pre.s:reproduce_audio() $\rightarrow \neg$ active_audio $\wedge \neg$ active_video;
pre.s:show_windows() $\rightarrow \neg$ active_audio $\wedge \neg$ active_video;
/* *Postconditions* */
post.u:set_volume(am) $\rightarrow$ (curr_volume = am *unless* *pre*.u:set_volume(am'));
post.u:redefine_window(g) $\rightarrow$ (curr_picture_size = g *unless* *post*.u:redefine_window(g'));
/* *Audio and video* */
s:reproduce_audio() $\rightarrow \bigcirc$ s:show_windows();

init:
curr_volume = empty_volume;
curr_picture_size = empty_size;

windows = empty_window_sequence;
$\neg$ active_conf; $\neg$ active_audio; $\neg$ active_video;
endinterface loc_conf;
/ Part of the system-interface related to audio-video conference */*
system-interface: i, j, k, n : *int*, ids: *sequ(int)*, pm: *protocol_message*, am: *audio*, g: *graphic*, ws: *window_sequ*;
/* n $\leq$ N *where* N *is the number of loc_struct interfaces* */
/ An audio and a video message are being sent by multicast to all participants: from the participant i the first time */*
loc_conf[i]: *post*.u:release_audio(am,g) $\rightarrow$ ($\wedge_{1\leq j\leq n, i=j}$ $\bigcirc_{loc\_conf[i]}$ (loc_istruct[j]: pm = insert_other_part_ids (insert_speaker_id (create_protocol_message(empty_protocol_message, am),i),list_of_part) $\wedge$ s:reproduce_audio(pm,am) $\wedge$ user_in_window_sequ(windows,i) = 0 $\wedge$ ws = windows) $\rightarrow$ $\bigcirc_{loc\_conf[j]}$ loc_conf[j]: (windows = add_window(ws, insert_user_name (insert_user_video(g, empty_window),i)) $\wedge$ show_windows(windows,g)));
/ An audio and a video message are being sent from the participant i to all participants not the first time */*
loc_conf[i]: *post*.u:release_audio(am,g) $\rightarrow$ ($\wedge_{1\leq j\leq n, i=j}$ $\bigcirc_{loc\_conf[i]}$ (loc_istruct[j]: pm = insert_other_part_ids (insert_speaker_id (create_protocol_message(empty_protocol_message, am),i),list_of_part) $\wedge$ s:reproduce_audio(pm,am) $\wedge$ user_in_window_sequ(windows,i) = k $\wedge$ k $\neq$ 0 $\wedge$ ws = windows) $\rightarrow$ $\bigcirc_{loc\_conf[j]}$ loc_conf[j]: (windows = replace_window(k,ws,insert_user_video (g, empty_window),i)) $\wedge$ show_windows(windows,g)));
/ Changes in the sequence of windows: the iniciator has closed the conference */*
loc_istruct[i]: ((active_conf $\wedge$ initiator $\wedge$ u:close_conf()) $\rightarrow$ $\wedge_{1\leq j\leq n}$ $\bigcirc_{loc\_conf[i]}$ loc_conf[j]: (windows = empty_window_sequ) $\wedge$ s:close_conf() $\wedge$ s:close_windows() $\wedge$ s:show_windows (empty_window_sequ))));
/ Changes in the sequence of windows: a participant i who is not the initiator has left the conference */*
loc_istruct[i]: ((active_conf $\wedge$ $\neg$ initiator $\wedge$ u:close_conf()) $\rightarrow$ $\wedge_{1\leq j\leq n, i\neq j}$ ($\bigcirc_{loc\_conf[i]}$ loc_conf[j]: (active_conf $\wedge$ ws = windows) $\rightarrow$ $\bigcirc_{loc\_conf[j]}$ (windows=remove_window(user_in_window_sequ(ws,i),ws) $\wedge$ s:close_conf()))) $\wedge$ windows = empty_window_sequ $\wedge$ show_windows(empty_window_sequ));
/ Only one participant may have the microphone activated at the same time (mutual exclusion) */*
$\wedge_{1\leq j\leq n}$ $\wedge_{1\leq i\leq n, i\neq j}$ (($\diamond_{loc\_conf[j]}$ loc_conf[j]: s:activate_audio() $\wedge$ $\diamond_{loc\_conf[j]}$ loc_conf[i]: s:activate_audio()) $\rightarrow$ ((loc_conf[i]: (s:release_audio() $\vee$ u:release_audio()) $before_{loc\_conf[j]}$ loc_conf[j]: s:activate_audio()) $\vee$ (loc_conf[j]: ((u:release_audio() $\vee$ s:release_audio()) $before_{loc\_conf[i]}$ loc_conf[i]: s:activate_audio())))));

The first two laws of the system-interface specification related to audio-video conference describe the situation where an audio message and the corresponding video are sent from by multicast from the speaker to all participants. If the message is the result of the user's first speach during the current conference, the speach is reproduced to all participants, and a window containing the speaker's name and video is inserted into the sequence of windows shown on the screen of all participants. If the speaker had already spoken before in the same conference, his latest speach is also reproduced to all participants. However, a new window need not to be opened for the speaker in the sequence of windows, but only a new video inserted into the existing window and shown to all participants. The operator $\bigcirc_{loc\_conf[i]}$ can together with the name and the index of the local istruct interface conveniently describe transmission by multicast in an abstract manner. Here we assume that the multicast is reliable. Since a conference is actually activated from istruct as described later in section 3.2.5, these laws are already a part of the system-interface specification related to istruct.

The next two laws of the system-interface specification specify the conferencing system behaviour when a participant leaves the conference. If the participant is the initiator, the conference is closed for all participants. Otherwise, the window of the leaving participant is removed from the window sequence and the updated sequence is shown to the remaining participants.

The last law specifies that only one conference participant may have the microphone activated at the same time (mutual exclusion). For the given formal specification of a conference is typical that not much functionality specified by sorts, but most of it by interfaces, especially by the system-interface. The reason is that a conference does not require complex data structures.

In the specification we did not describe:

- The video in a window is considered as a whole, imported from the sort *graphic*. Therefore, it is not divided into parts specifying movements of users and their mental behaviour. After an analysis of the relevant movements and behaviour, an additional set of actions could be defined and included into the specification. The set should have a finite and reasonably small number of elements.

- We assumed that handing the initiatory over to another user is not possible.

- Continuous transmission of audio and video signals has not been represented for the reasons already explained at the beginning of this subsection.

Involvement of a social protocol

An audio-video conference is one of the typical CSCW applications, where cooperation between users is partly controlled by a social protocol. This is a difference to "classical" distributed systems, since the effect of some social protocol rules has to be taken into account also in the formal specification of a CSCW application.

In our opinion, it is better not to integrate the specification of social protocol rules directly into the specification of the system. These are actually two layers of interaction which should be described separately. A specification of the social protocol rules related to an application should be done first. Knowing its contents, the application can be formally described.

In an audio-video conference, there is possible that a user has his microphone activated after execution of the actions "u:want_to_speak" and "s:activate_audio". However, if he/she for some reason does not start to speak or the contents of the speach, its length or behaviour while talking become unacceptable, the chairman has to a have a possibility to deactivate his/her microphone. For this purpose, the actions "u:take_audio" and "s:release_audio" are foreseen. The following social protocol rules should be known while specifying the conference, but not included in the interface specification:

user[i]: u:no_talk() $\rightarrow \bigcirc_{user[i]}$ (user[j]: initiator $\rightarrow$ u:take_audio());
u:no_talk() $\rightarrow$ s:release_audio();

As shown in this short example, the temporal logic can also be used to specify social protocol rules.

3.2.4 Specification of the interfaces for awareness tools

Awareness tools are used to support group aware collaboration. The Iris environment includes the following awareness tools [Iris96]:

(a) iuserinfo - Iris Session Info Tool for displaying the session information (user "list"),

(b) ihostinfo - Iris Host Info Tool for displaying the host status ("list" of hosts participating in the session), and

(c) idocinfo - Iris Document Info Tool for presentation of the information on the currently edited documents.

For each tool we give a short informal description and a formal specification of the interface and accompanied sorts.

Interface for iuserinfo

iuserinfo displays the users who are currently working or worked on a document. The users represented by icons are divided into the groups of active and passive users. All the icons are supposed to be shown on the screen at the same time. For each user from any group, the following information is available:

- name (username),
- picture,
- role (author/reviewer),

- current status (online/offline), and
- general status (on leave/other additional information on status).

Since the ordering of the icons inside the groups is for presentation in istruct not particularly important, we represent the list of icons by a set and define the sort *set_of_users*. The set consists of two subsets representing the groups of active and passive users described by two elements of the sort *subset_of_users*. Each icon is an element of the sort *user*.

We specify first the sort *user* very similar to the sort *window* since all operations providing the information on a user are performed by the system. We will also not explain the functionality of each operation in detail (all insert-operations are used to insert different parts of information on a user and all get-operations return them).

sort *user*: empty_user, insert_username, insert_picture, insert_role, insert_curr_status, insert_gen_status, get_username, get_picture, get_role, get_curr_status, get_gen_status;
based on *int*, *graphic*, *string*;
opns
  empty_user: *user*;
  insert_username(u: *user*, username: *int*): *user*;
  insert_picture(u: *user*, pic: *graphic*): *user*;
  insert_role(u: *user*, role: *string*): *user*;
  insert_curr_status(u: *user*, cs: *string*): *user*;
  insert_gen_status(u: *user*, gs: *string*): *user*;
  get_username(u: *user*): *int*;
  get_picture(u: *user*): *graphic*;
  get_role(u: *user*): *string*;
  get_curr_status(u: *user*): *string*;
  get_gen_status(u: *user*): *string*;
laws u: *user*, username: *int*, pic: *graphic*, role, cs, gs: *string*;
  get_username(empty_user) = 0;
  get_picture(empty_user) = empty_picture;
  get_role(empty_user) = empty_string;
  get_curr_status(empty_user) = empty_string;
  get_gen_status(empty_user) = empty_string;
  get_username(insert_username(u,username)) = username;
  get_username(insert_picture(u,pic)) = username;
  get_username(insert_role(u,role)) = get_username(u);
  get_username(insert_curr_status(u,cs)) = get_username(u);
  get_username(insert_gen_status(u,gs)) = get_username(u);
  get_picture(insert_picture(u,pic)) = pic;
  get_role(insert_role(u,role)) = role;
  get_curr_status(insert_curr_status(u,cs)) = cs;

get_gen_status(insert_gen_status(u,gs)) = gs;
/* All the get-operations return the requested parameter value independent of the order in which other parameter values have been inserted (as we completely specified for the operation "get_username" */
endsort *user*;

In the following we specify a sort describing some elementary operations on a set which will later be needed for specification of the istruct interface. An element of the set may be of any other sort which we for generality call *element*. This sort also contains the operation "empty_element". The operations "insert" and "remove", in that order, insert a new element into a set and remove an existing element from the set. "is_in_set(s,e)" is "true" if the element "e" is in the set "s". The sort *set* is a generic sort.

sort *set* (*element*[empty_element]): empty_set, insert_element, remove_element, is_in_set;
based on *bool*;
opns
empty_set: *set*;
insert_element(s: *set*, e: *element*): *set*;
remove_element(s: *set*, e: *element*): *set*;
get_element(s: *set*, e: *element*): *element*;
is_in_set(s: *set*, e: *element*): *bool*;
laws s: *set*, e,ex,ey: *bool*;
insert(insert(s,e),e) = insert(s,e);
insert(insert(s,ex),ey) = insert(insert(s,ey),ex);
remove(insert(s,e),e) = s;
remove(insert(s,ex),ey) = insert(remove(s,ey),ex);
remove(empty_set,e) = empty_set;
is_in_set(insert(s,e),e);
$\neg$ is_in_set(empty_set,e);
ex $\neq$ ey $\rightarrow$ is_in_set(insert(s,ex),ey) = is_in_set(s,ey);
endsort *set*;

Several sorts used in the specification of interfaces for awareness tools have the same operations and laws as the sort *set*. The names of the sorts and elements are as specified in Table 3.1.

Next we specify the sort *set_of_users*. It consists of two disjoint subsets: the first is an element of the sort *active_users* and the second is an element of *passive_users*. One of the subsets may be empty.

sort *set_of_users*: construct_set_of_users, get_subset_of active, get_subset_of_passive;
based on *subset_of_users*, *bool*;

interface	name of sort	instantiation
iuserinfo	*subset_of_users*	*set*(*user*[empty_user])
ihostinfo	*set_of_hosts*	*set*(*host*[empty_host])
idocinfo	*set_of_documents*	*set*(*doc_info*[empty_doc_info])
idocinfo	*set_of_replicas*	*set*(*replica*[empty_replica])
idocinfo	*set_of_curr_vers*	*set*(*version*[empty_version])
idocinfo	*set_of_acc_rights*	*set*(*access_right*[0])

Table 3.1: Use of the sort *set*

opns
create_set_of_users(sa: *subset_of_users*, sp: *subset_of_users*): *set_of_users*;
get_subset_of_active(su: *set_of_users*): *subset_of_users*;
get_subset_of_passive(su: *set_of_users*): *subset_of_users*;
laws su: *set_of_users*, sa,sp: *subset_of_users*, u: *user*;
get_subset_of_active(construct_set_of_users(sa,sp)) = sa;
get_subset_of_passive(create_set_of_users(sa,sp)) = sp;
get_subset_of_active(construct_set_of_users(sa,sp)) = get_subset_of_active(su);
get_subset_of_passive(construct_set_of_users(sa,sp)) = get_subset_of_passive(su);
is_in_set(get_subset_of_active(su),u) → ¬ is_in_set(get_subset_of_passive(su),u);
get_subset_of_active(su) = empty_set → get_subset_of_passive(su) ≠ empty_set;
get_subset_of_passive(su) = empty_set → get_subset_of_active(su) ≠ empty_set;
endsort *set_of_users*;

iuserinfo is activated and closed from istruct by the system actions "s:open_user_info" and "s:close_user_info". After activation or if any information on a user has been changed, the set of users represented by icons is updated by the system action "s:update_users(su)" where the parameter "su" denotes the set of users.

Besides the system actions, there are also user actions available in loc_user_info. "u:select_users(sel)" is the action to select a user (users) for sending an email or for an initiation of a conference. The action returns a sequence "sel" of username(s) of the selected user(s). "u:insert_gen_status(gs)" provides a user the possibility to enter a string "gs" containing information on his/her general status.

interface loc_user_info: s:open_user_info, s:close_user_info, s:update_users, users, s:logged_in, s:logged_out, u:select_users, u:insert_gen_status;
based on *set_of_users*;
instances
users: *set_of_users*;
general_status: *string*;

actions:

s:open_user_info(), s:close_user_info(), s:update_users(su: *set_of_users*), u:select_users(sel: *sequ(int)*), u:insert_gen_status(gs: *string*), logged_in(h: *host*), s:logged_out(h: *host*);

laws: su: *set_of_users*, u: *user*;

/* *Postconditions* */

s:open_user_info() → (DISPLAY_SET(users) *until* s:close_user_info());

s:update_users(su) → (users = su *atnext post*.s:update_users(su));

u:insert_gen_status(gs) → (general_status = gs *atnext post*.u:insert_gen_status(gs));

post.s:logged_out(h) → s:close_user_info();

/* *Only one iuserinfo active at the same time* */

s:open_user_info() → (¬ s:open_user_info() *until* s:close_user_info());

start ∨ s:close_user_info() → (s:open_user_info() *before* (s:update_users() ∨ u:select_users() ∨ u:insert_gen_status() ∨ s:close_user_info()));

init:

users = empty_set; general_status = empty_string;

endinterface loc_user_info;

/* *Part of the system-interface related to iuserinfo* */

system-interface: i,j,n: *int*, su: *set_of_users*, u: *user*;

/* *Similar to the laws of loc_host_info, due to the similarity they will not be specified* */

Interface for ihostinfo

ihostinfo displays all hosts involved in editing one or more documents. For each host, an icon with the name and the list of users (usernames) is shown. The icons can be represented by a set which is an element of the sort *set_of_hosts* where each host is an element of the sort *host*. *host* is based on the sort *int* for names and on *sequ(int)* for the list of users. Since its operations are similar to the operations of the sort *user*, it needs no further explanation.

sort *host*: empty_host, insert_host_name, insert_list_of_users, get_host_name, get_list_of_users;

based on *int*, *sequ(int)*;

opns

empty_host: *host*;

insert_host_name(h: *host*, hn: *int*): *host*;

insert_list_of_users(h: *host*, ul: *sequ(int)*): *host*;

get_host_name(h: *host*): *int*;

get_list_of_users(h: *host*): *sequ(int)*;

laws h: *host*, hn: *int*, ul: *sequ(int)*;

get_host_name(empty_host) = 0;

get_list_of_users(empty_host) = empty_sequ_int;
get_host_name(insert_host_name(h,hn)) = hn;
get_list_of_users(insert_list_of_users(h,ul)) = ul;
get_host_name(insert_list_of_users(h,ul)) = get_host_name(h);
get_list_of_users(insert_host_name(h,hn)) = get_list_of_users(h);

endsort *host*;

Like the other two awareness tools, ihostinfo is called automatically after activation of istruct and closed together with istruct. It is opened by the action "s:open_host_info" and closed by "s:close_host_info". After ihostinfo is activated, a set of icons is updated as specified by the system action "s:update_hosts(sh)" and displayed to a user. The parameter "sh" denotes a set of hosts represented by the icons. Functionality of popup-menus which are used to provide some additional information on hosts and to reorder icons in a window will not be specified, since it is not necessarily needed for specification of the istruct interface. The specification of the loc_host_info interface therefore contains no user actions. ihostinfo only shows the contents of data structures provided exclusively by the system.

The data described by an element of *set_of_hosts* and its element from *host* are changed only when a participant leaves the session or a new participant joins the session. The changes can be specified on the system level only. If a user has logged in/out, his name has to be added to/removed from the list of users for that particular host, and the updated information on the host (represented by an element of *host*) has to be changed in the set of hosts.

We assume that ihostinfo gets the required information on hosts from the operating system by the system actions "s:logged_out(h)" and "s:logged_in(h)". The system actions are the result of the user actions "u:logout" and "u:login" executed during the interaction of a user with the operating system. The parameter "h" represents the host where a user is logged in/out.

The operations "insert_el_in_sequ_int" and "remove_el_from_sequ_int" in the following specification of the loc_host_info interface are assumed to be defined in the sort *sequ(int)*. "DISPLAY_SET(hosts)" is a variable which specifies the actual presentation of a set of hosts represented by icons to a user (displaying the complete set of icons in a window).

interface loc_host_info: s:open_host_info, s:close_host_info, s:update_hosts, hosts, s:logged_in, s:logged_out;

based on *set_of_hosts*;

instances

hosts: *set_of_hosts*;

actions:

s:open_host_info(), s:close_host_info(), s:update_hosts(sh: *set_of_hosts*), s:logged_in(h: *host*), s:logged_out(h: *host*);

laws: sh: *set_of_hosts*, h: *host*;

/* *Postconditions* */
s:open_host_info() $\rightarrow$ (DISPLAY_SET(hosts) *until* s:close_host_info());
post.s:update_hosts(sh) $\rightarrow$ hosts = sh;
post.s:logged_out(h) $\rightarrow \bigcirc$ s:close_host_info();
/* *Only one ihostinfo active at the same time* */
s:open_host_info() $\rightarrow$ ($\neg$ s:open_host_info() *until* s:close_host_info());
start $\vee$ s:close_host_info() $\rightarrow$ (s:open_host_info() *before* (s:update_hosts() $\vee$ s:close_host-_info()));

init:

hosts = empty_set;

endinterface loc_host_info;

/* *Part of the system-interface related to ihostinfo* */

system-interface: i,j,n: *int*, sh: *set_of_hosts*, h: *host*;

/* *A participant i who had worked on the host h has logged out* */

loc_host_info[i]: (sh = hosts $\wedge$ s:logged_out(h)) $\rightarrow \wedge_{1\leq j\leq n, i\neq j} \bigcirc_{loc\_host\_info[i]}$ (loc_host_info[j]: update_hosts (insert(remove(sh,h), insert_list_of_users(h, remove_el_from_sequ_int (get_list_of_users(h),i))));

/* *Update of information on the participant i for the next login* */

loc_host_info[i]: ((s:logged_out(h) $\wedge$ hosts = sh) $\rightarrow$ (hosts = insert(remove(sh,h), insert_list_of_users(h, remove_el_from_sequ_int(get_list_of_users(h),i))) *atnext* *post*.s:logged-_out(h)));

loc_host_info[i]: ((s:logged_in(h) $\wedge$ hosts = sh) $\rightarrow$ (hosts = insert(sh,insert_list_of_users(h, insert_el_in_sequ_int(get_list_of_users(h),i)))));

/* *A new participant i has logged in on the host h* */

loc_host_info[i]: (sh = hosts $\wedge$ s:logged_in(h)) $\rightarrow \wedge_{1\leq j\leq n, i=j} \bigcirc_{loc\_host\_info[i]}$ (loc_host_info[j]: update_hosts (insert(sh,insert_list_of_users (h, insert_el_in_sequ_int(get_list_of_users(h),i)))));

Interface for idocinfo

idocinfo displays for each document:

- its owner,
- a list of replicas,
- a list of current versions, and
- a list of users with different access rights.

The list of replicas, the list of current versions and the list of users with different access rights are represented by elements of the sort *set_of_replicas* based on the sort *replica*, *set_of_curr_vers* based on *version* and *set_of_acc_rights* based on *access_right*. *replica*, *version* and *access_right* are similar to *user* or *host* and therefore will not be specified in detail. We give only a list of their operations:

- *replica:*
  - empty_replica: *replica*;
  - insert_host_name(re: *replica*, hn: *int*): *replica*;
  - insert_time(re: *replica*, time: *int*): *replica*;
  - get_host_name(re: *replica*): *int*;
  - get_time(re: *replica*): *int*;
- *version:*
  - empty_version: *version*;
  - insert_username(vs: *version*, username: *int*): *version*;
  - insert_host_name(vs: *version*, hn: *int*): *version*;
  - insert_version_number(vs: *version*, version_no: *int*): *version*;
  - get_username(vs: *version*): *int*;
  - get_host_name(vs: *version*): *int*;
  - get_version_number(vs: *version*): *int*;
- *access_right:*
  - empty_access_right: *access_right*;
  - insert_username(ar: *access_right*, username: *int*): *access_right*;
  - insert_right(ar: *access_right*, right: *string*): *access_right*;
  - get_username(ar: *access_right*): *int*;
  - get_right(ar: *access_right*): *string*;

Only information on one document is shown at the same time. Information on each document can be represented by an element of the set *set_of_documents*.

sort *doc_info*: empty_doc_info, insert_owner, insert_set_of_replicas, insert_set_of_curr_vers, insert_set_of_acc_rights;
based on *int*, *set_of_replicas*, *set_of_curr_vers*, *set_of_acc_rights*;
opns
empty_doc_info: *doc_info*;
insert_owner(di: *doc_info*, ow: *int*): *user*;
insert_set_of_replicas(di: *doc_info*, sr: *set_of_replicas*): *doc_info*;
insert_set_of_curr_vers(di: *doc_info*, scv: *set_of_curr_vers*): *doc_info*;

insert_set_of_acc_rights(di: *doc_info*, sar: *set_of_acc_rights*): *doc_info*;
get_owner(di: *doc_info*): *int*;
get_set_of_replicas(di: *doc_info*): *set_of_replicas*;
get_set_of_curr_vers(di: *doc_info*): *set_of_curr_vers*;
get_set_of_acc_rights(di: *doc_info*): *set_of_acc_rights*;
laws di: *doc_info*, ow: *int*, sr: *set_of_replicas*, scv: *set_of_curr_vers*, sar: *set_of_acc_rights*;
get_owner(empty_doc_info) = 0;
get_set_of_replicas(empty_doc_info) = empty_set;
get_set_of_curr_vers(empty_doc_info) = empty_set;
get_set_of_acc_rights(empty_doc_info) = empty_set;
get_owner(insert_owner(di,ow)) = ow;
get_owner(insert_set_of_replicas(di,sr)) = get_owner(di);
get_owner(insert_set_of_curr_vers(di,scv)) = get_owner(di);
get_owner(insert_set_of_acc_rights(di,sar)) = get_owner(di);
get_set_of_replicas(insert_set_of_replicas(di,sr)) = sr;
get_set_of_curr_vers(insert_set_of_curr_vers(di,scv)) = scv;
get_set_of_acc_rights(insert_set_of_acc_rights(di,sar)) = sar;
/* All the get-operations return the requested parameter value independent of the order in which other parameter values have been inserted (as we completely specified for the operation "get_owner" */
endsort *doc_info*;

Actions of the idocinfo interface are system actions displaying only the data specified by the related sorts. The interface is very similar to the iuserinfo and ihostinfo interfaces and therefore will not be specified.

In the specification of interfaces for awareness tools can be seen how a tool functionality reflects in the contents of the interface specification. Since the functionality of the awareness tools is mostly to update and show the contents of data structures, the major part of the functionality is specified by operations in sorts. CADT have been proved to be quite convenient for this purpose.

3.2.5 Specification of the istruct interface

A structure object edited in istruct can in general be defined as an n-tree containing nodes of two types:

- *Structure nodes* are used for structure definition of a document. All nodes in a tree except leaves are structure nodes.
- *Document nodes* are used as pointers to contents objects (real documents). Only leaves of a tree are document nodes.

Contents objects are assigned different document types. A Mimetype of a document may be an ASCII text, HTML pages or another format defined in RFC 1521 and 1522 (for the complete list of types supported in Iris see [Koch97]).

We specify a structured object by the sort with the same name. Since a structured object is composed of nodes, the sort *structured_object* is based on the sort *struct_obj_node*.

struct_object_node defines operations on a node of a structured object. It is based on elementary sorts *nat* (natural numbers), *sequ(nat)* (sequence of natural numbers) and *bool* (boolean) which are often used in algebraic specifications and will not be defined in this report. An interested reader may refer to [Wirs90] for details.

In the following definition of *struct_obj_node* we also use the sorts *structured_object* and *document*. "set_lock" and "delete_lock" are operations with signature (sequ(nat), structured_object): structured_object in the sort *structured_object*. The sort *document* is assumed to already contain an operation of assignment of a Mimetype to a particular document. "empty_string" is assumed to be defined in the sort *string*.

The operations "add_succ" and "delete_succ" add and delete the n-th successor of a node. "get_succ" returns the n-th successor of a node and "get_number_succ" the number of its successors. "make_document_node", "is_document_node" and "get_document" are operations related to document nodes. "make_document_node" creates a node with the property of a document node, "is_document_node" returns "true" if a node has that property and "get_document" returns the document where the node is pointing to. "lock_node", "delete_lock_node" and "is_locked_node" are used together with the lock-operations from the sort *structured_object* to describe explicit and implicit locking of nodes in istruct. "set_string" allows to insert a new string or replace an existing string assigned to a node and "get_string" returns its contents. "compare_struct_obj_nodes" is "true" if the compared nodes are equal. "compare_documents" comparing contents of two files is assumed to be defined in the sort *file*.

sort *struct_obj_node*: empty_struct_obj_node, add_succ, get_number_succ, delete_succ, get_succ, make_document_node, get_document, is_document_node, lock_node, delete_lock_node, is_locked_node;

based on *structured_object*, *nat*, *sequ(nat)*, *bool*, *document*;

opns

empty_struct_obj_node: *struct_obj_node*;
add_succ(node: *struct_obj_node*, n: *nat*, t: *structured_object*): *struct_obj_node*;
get_number_succ(node: *struct_obj_node*): *nat*;
delete_succ(node: *struct_obj_node*, n: *nat*): *struct_obj_node*;
get_succ(node: *struct_obj_node*, n: *nat*): *structured_object*;
make_document_node(node: *struct_obj_node*, doc: *file*): *struct_obj_node*;
get_document(node: *struct_obj_node*): *file*;
is_document_node(node: *struct_obj_node*): *bool*;
lock_node(node: *struct_obj_node*): *struct_obj_node*;
delete_lock_node(node: *struct_obj_node*): *struct_obj_node*;

is_locked_node(node: *struct_obj_node*): *bool*;
set_string(node: *struct_obj_node*, s: *string*): *struct_obj_node*;
get_string(node: *struct_obj_node*): *string*;
compare_struct_obj_nodes(node1: *struct_obj_node*, node2:*struct_obj_node*): *bool*;

laws node, node1, node2: *struct_obj_node*, n,m: *nat*, t: *structured_object*, doc: *document*, s: *string*;

get_number_succ(empty_struct_obj_node) = 0;
get_number_succ(node) = n-1 → get_number_succ(add_succ(node,n,t)) = n;
get_number_succ(node) ≠ n-1 → add_succ(node,n,t) = node;
n > m → get_succ(add_succ(node,n,t),m) = get_succ(node,m);
get_succ(add_succ(node,n,t),n) = t;
n > get_number_succ(node) → get_succ(node,n) = empty_structured_object;
n > m → delete_succ(add_succ(node,n,t),m) = delete_succ(node,m);
delete_succ(add_succ(node,n,t),n) = node;
n > get_number_succ(node) → add_succ(delete_succ(node,n),n-1,t) = node;
delete_succ(make_document_node(node,doc),n) = delete_succ(node,n);
get_number_succ(node) > 0 → ¬ is_document_node(node);
get_number_succ(node) = 0 → is_document_node(make_document_node(node,doc));
is_document_node(node) → get_document(make_document_node(node,doc)) = doc;
is_document_node(node) → add_succ(node,n,t) = node;
is_document_node(node) → is_document_node(add_succ(node,n,t));
is_document_node(add_succ(node,n,t)) → get_document(add_succ(node,n,t)) = get_document(node);
is_document_node(delete_succ(node,n)) → get_document(delete_succ(node,n)) = get_document(node);
¬ is_document_node(node) → get_document(node) = empty_document;
¬ is_document_node(node) → ¬ is_document_node(add_succ(node,n,t));
¬ is_document_node(make_document_node(node,doc)) → add_succ(make_document_node(node,doc),n,t) = add_succ(node,n,t);
lock_node(add_succ(node,n,t)) = add_succ(lock_node(node),n,set_lock(empty_sequ_nat, t));
¬ is_locked_node(empty_struct_obj_node);
is_locked_node(lock_node(empty_struct_obj_node));
delete_lock_node(lock_node(node)) = node;
delete_lock_node(empty_struct_obj_node) = empty_struct_obj_node;
delete_lock_node(add_succ(node,n,t)) = add_succ(delete_lock_node(node), n, delete_lock(empty_sequ_nat,t));
delete_lock_node(lock_node(node),n) = lock_node(delete_lock_node(node,n));
get_string(empty_struct_obj_node) = empty_string;
get_string(set_string(node, s)) = s;
get_string(add_succ(node, n)) = get_string(node);
get_string(delete_succ(node, n)) = get_string(node);

get_string(delete_lock_node(node)) = get_string(node);
get_string(make_document_node(node,doc)) = get_string(node);
get_string(lock_node(node)) = get_string(node);
compare_struct_obj_nodes(empty_struct_obj_node, empty_struct_obj_node);
node1 $\neq$ empty_struct_obj_node $\rightarrow \neg$ compare_struct_obj_nodes(node1, empty_struct-_obj_node);
node2 $\neq$ empty_struct_obj_node $\rightarrow \neg$ compare_struct_obj_nodes(empty_struct_obj_node, node2);
node1 $\neq$ empty_struct_obj_node $\wedge$ node2 $\neq$ empty_struct_obj_node $\rightarrow$ compare_struct-_obj_nodes(node1, node2) = compare_struct_obj_nodes(get_succ(node1, get_number_succ (node1)), get_succ(node2, get_number_succ(node2))) $\wedge$ compare_struct_obj_nodes (delete_succ(node1, get_num_succ(node1)), delete_succ(node2, get_num_succ(node2))) $\wedge$ get_string(node1) = get_string(node2) $\wedge$ (is_document_node(node1) $\leftrightarrow$ is_document_node(node2) $\wedge$ compare_documents(get_document(node1), get_document(node2)));
endsort *struct_obj_node*;

The sort *structured_object* defined next contains similar groups of operations:

- operations for addition of a node to a structure object ("add_struct_obj_node", "insert_struct_object_node"), and deletion of a node ("delete_struct_obj_node"),
- operations for locking ("set_lock", "delete_lock", "is_locked"),
- the operation for comparison of structure objects "compare", and in addition
- copy and move operations on subtrees ("copy_subtree" and "move_subtree").

While "add_struct_obj_node" abstractly describes only an addition of a node to a structure object, "insert_struct_obj_node" specifies its actual insertion onto the position defined by the sequence "ns" of node successor indices in the path from the root to the current node and by the index "n" of the successor node which we want to insert as the n-th successor of the current node. The assignment of the successor indices 1,...,n always starts with the left most successor.

"move_subtree" and "copy_subtree" are defined using the operation "insert_struct_object_node" for each node in the subtree. In "move_subtree", the subtree on the original position is after that deleted. n_1 and ns_1 in both operations define the original position and n_2 and ns_2 the new position of the moved/copied subtree.

A complete specification of locks and intention locks of istruct is in this sort (and also in the local istruct interface) impossible, because it requires global information on other users working on the document which is known only on the level of the system-interface.

"first" and "rest" are operations in the sort *sequ(nat)*. In the sort *sequ(nat)*, there is also an operation "postfix" with the signature (sequ(nat),nat): sequ(nat).

sort *structured_object*: empty_structured_object, add_struct_obj_node, delete_struct_obj-_node, select_node, get_node, insert_struct_obj_node, copy_subtree, move_subtree, set_lock, delete_lock, is_locked;

based on *struct_obj_node*, *bool*, *nat*, *sequ(nat)*, *document*;

opns

empty_structured_object: *structured_object*;
add_struct_obj_node(node: *struct_obj_node*, t: *structured_object*): *structured_object*;
delete_struct_obj_node(ns: *sequ(nat)*, n: *nat*, t: *structured_object*): *structured_object*;
select_node(ns: *sequ(nat)*, t: *structured_object*): *struct_obj_node*;
get_node(t: *structured_object*): *node*;
insert_struct_obj_node(ns: *sequ(nat)*, n: *nat*, node: *struct_obj_node*, t: *structured_object*): *structured_object*;
copy_subtree(ns_1: *sequ(nat)*, n_1: *nat*, ns_2: *sequ(nat)*, n_2: *nat*, t: *structured_object*): *structured_object*;
move_subtree(ns_1: *sequ(nat)*, n_1: *nat*, ns_2: *sequ(nat)*, n_2: *nat*, t: *structured_object*): *structured_object*;
set_lock(ns: *sequ(nat), t:* structured_object): *structured_object*;
delete_lock(ns: *sequ(nat)*, t: *structured_object*): *structured_object*;
is_locked(ns: *sequ(nat)*, t: *structured_object*): *bool*;
compare(t1: *structured_object*, t2: *structured_object*): *bool*;

laws ns,ns_1,ns_2: *sequ(nat)*, n,n_1,n_2: *nat*, t, t1, t2: *structured_object*, node: *struct_obj_node*;

get_node(empty_structured_object) = empty_struct_obj_node;
get_node(add_node(node,t)) = node;
insert_struct_obj_node(empty_sequ_nat,n,node,t) = add_struct_obj_node(add_succ(get_-node(t), n, add_struct_obj_node(node, empty_structured_object)), empty_structured-_object);
¬ isempty(ns) → insert_struct_obj_node(ns,n,node,t) = add_struct_obj_node(add_succ (delete_succ(get_node(t), first(ns)), first(n), insert_struct_obj_node(rest(ns), n, node, get_succ(get_node(t), first(ns)))), empty_structured_object);
select_node(empty_sequ_nat,t) = get_node(t);
¬ isempty(ns) → select_node(ns,t) = select_node(rest(ns),get_succ(get_node(t), first(ns)));
copy_subtree(ns_1,n_1,ns_2,n_2,t) = insert_struct_obj_node(ns_2, n_2,select_node(postfix(ns_1, n_1),t),t);
move_subtree(ns_1,n_1,ns_2,n_2,t) = delete_struct_obj_node(postfix(ns_1,n_1), n_1, insert_struct_obj_node(ns_2, n_2,select_node(postfix(ns_1,n_1), t), t));
set_lock(empty_sequ_nat,t) = add_struct_obj_node (lock_node(get_node(t)), empty_structured_object);
¬ isempty(ns) → set_lock(ns,t) = add_struct_obj_node(add_child(delete_succ(get_node(t), first(ns)),set_lock(rest(ns), get_succ(get_node(t), first(ns)))), empty_structured_object);
is_locked(empty_sequ_nat,t) = is_locked_node(get_node(t));
¬ isempty(ns) → is_locked(ns,t) = is_locked(rest(ns),get_succ(get_node(t),first(ns)));

delete_struct_obj_node(empty_sequ_nat,n,t) = add_struct_obj_node(delete_succ(get_node(t),n), empty_structured_object);
$\neg$ isempty(ns) $\rightarrow$ delete_struct_obj_node(ns,n,t) = add_struct_obj_node(add_succ(delete_succ(get_node(t), first(ns)), first(ns), delete_struct_obj_node(rest(ns), n, get_succ(get_ node(t), first(ns)))), empty_structured_object);
compare(empty_structured_object, empty_structured_object);
t1 $\neq$ empty_structured_object $\rightarrow \neg$ compare(t1, empty_structured_object);
t2 $\neq$ empty_structured_object $\rightarrow \neg$ compare(empty_structured_object, t2);
t1 $\neq$ empty_structured_object $\wedge$ t2 $\neq$ empty_structured_object $\rightarrow$ compare(t1,t2) = compare_struct_obj_nodes(get_node(t1),get_node(t2));

endsort *structured_object*;

Since the definition of sorts and interfaces included in istruct is given, it is possible to precisely specify the loc_istruct compound interface. Actions are related to the functionality of istruct as a structure editor or to its interaction with external applications (email, audio-video conference and awareness tools). First, the istruct functionality visible in its external behaviour has to be identified and after that the sketch of a formal specification of loc_istruct given in subsection 3.2.1 has to be refined.

Following the Iris informal descriptions in [Koch95a, Koch95b, Iris96, Koch97] and some other informal descriptions of MBone tools and electronic mailers (especially manuals), we identify the following parts of the istruct functionality which affect its external behaviour and should be described in the interface specification:

(a) *Editing structured objects without interaction with other applications.* The functionality includes:

- The start of istruct on a particular structure object and closing istruct.

  This can be specified by the user actions "u:open_istruct(t)" and "u:close_istruct" where the parameter t of the sort *structured_object* denotes the structured object.

- Creation and changing structure views. A structure view is a window showing either the complete structure of nodes (a tree) for the currently edited structured object or only a part of it (a subtree). The default structure view is the view of the root node only and is opened after starting istruct without selection of any node. New views may after that be created by selection of a node in the tree and by disabling the identificators of its successors. These identificators are required to be shown to a user before enabling/disabling. In this way only the desired part of the tree may be viewed and the others hidden. The hidden nodes cannot be selected in that view any more. A structure view may be closed any time. Closing the last view, the editor is closed.

  We specify selection of a node by the user action "u:sel_node(ns)" where the parameter "ns" of sort *sequ(nat)* represents the position of the selected node. Although in a different view, each new structure view actually represents the same element of *structured_object*. Therefore we specify only one structure view for one structured object.

- Changing the object structure. It is possible to insert or delete a node, to copy or move a subtree, and to update the node attributes. The node attributes are the name (i.e. a document title or a title of a subdocument containing a chapter, section, etc.) and the identifier of the document type (i.e. Mimetype). The latter are specified only for leaves of a tree. All the structure editing operations implicitly lock the involved nodes of a structured object. The nodes become unlocked again as soon as an execution of an structure-changing operation is finished. A new version of the document is distributed to all users. The structure-changing operations are initiated by selecting nodes in the current structure view followed by selection of an operation.

  The related structure-changing user actions in the specification are "u:insert_node(node: *struct_obj_node*)", "u:del_node(n: *nat*)", "u:copy_subtr(n1: *nat*, ns2: *sequ(nat)*, n2: *nat*)", "u:move_subtr(n1: *nat*, ns2: *sequ(nat)*, n2: *nat*)" and "u:update_attr(s: *string*)". All the actions are based on the corresponding operations of *structured_object*. A new version of a document is distributed to other users by the system action "s:update_version(t: *structured_object*)".

- Explicit locking. It is possible to lock a node explicitly if it is selected and not already locked. Otherwise, the attempt of locking is unsuccessful.

  An explicit lock is required for the currently selected node by the user action "u:intend_to_lock". If allowed, the lock is done by the system action "s:expl_lock(ns: *sequ(nat)*)". Otherwise, the user gets a rejection by "s:display_lock_not_possible". A node becomes unlocked again by the action "s:end_lock".

- Locked nodes and nodes with intention locks (the latter are nodes which cannot be locked because their successors are already locked) are to be displayed in different colours than the unlocked nodes. The colour of the node is the only information to a user about the status of the node (locked, unlocked, under intention lock). This information cannot be displayed without interaction with the idocinfo awareness tool. A user attempt to lock the nodes coloured as locked or intentionally locked is always unsuccessful.

  An implicit lock is done by the system action "s:impl_lock(ns: *sequ(nat)*)" and an intention lock with "s:int_lock(node,t,st)".

(b) *Editing including interaction with a content editor*:

- Start of a content editor. The content editor is an application external to istruct (usually a text editor or an editor for other document types) which is invoked to edit the contents of a document. An appropriate editor for the document type is started automatically by selecting a node pointing to a document.

The described is performed upon the user action "u:sel_for_editing(ns: *sequ (nat)*)".

(c) *Interaction with awareness tools.*

- Start of the user_info, host_info and doc_info tools. The awareness tools are started automatically after the start of istruct.

  The start of awareness tools is specified by the system actions "s:open_user_info", "s:open_host_info" and "open_doc_info" already defined in their interfaces.

- Resolution of conflicts related to different versions of replicated documents. By the comparison between the remote and the local history in idocinfo, the conflicting updates may be determined. Despite that eventual conflicts have to be notified, it must be possible to continue work with the conflicting versions.

  Specifying that functionality we assume that a new version is created after each "save" in a local text editor while editing a document or after any structure changing operation on the document. Replicas are copies of the document of the same version. A conflict means a difference in contents or structure between two versions of the document.

  The action "s:conflict_notif()" is a notification on conflict referred to different versions of a document.

(d) *Interaction with an audio-video conference.*

- A conference may be initiated (as described below, an initiation is made through the interaction with the iuserinfo tool) by an initiator. We assume here that, if a conference already exists, another one must not be initiated inside the same set of users or its subset until the first conference is closed.

- An invitation to a conference may be or may not be accepted by the invited user. If all invited users have rejected the invitation, the conference cannot be established.

  The action of invitation is "s:invite_to_conf" and the user actions of acceptance/rejection are "u:join_conf" and "u:not_join_conf".

- Under certain circumstances, the right to speak may be taken the current speaker away by the initiator. This part of functionality refers to a social protocol as already discussed at the end of the previous subsection.

  For taking floor resp. the microphone, the initiator may use the action "u:take_audio". This can be specified in the system-interface only, since the identity of the initiator is in the lower-level interfaces unknown.

(e) *Interaction with email.*

- A notification of acceptance of new mails with a subject containing the name of the edited structured object has to be provided.

- The possibility to start an electronic mailer is needed (iuserinfo is involved here again as described in the next item).

(f) *Interaction with several external applications at the same time.*

- Sending an email message may be initiated from istruct by clicking a user icon in iuserinfo. A message may be sent to one from the set of users who are working or worked in the past on the currently edited document. By clicking one or more user icons, identification of recipients is provided for istruct. The identification of recipients and the subject (name of the document) are inserted automatically. If the subject contains the name of a structured object, the recipient will automatically get a notification on arrival of the message. There is no possibility of calling the mail interface directly by a user.
- An audio-video conference may also be initiated by clicking user icons in iuserinfo. This is possible for users only whose current status in iuserinfo is "online".

  The related actions of initiation are "u:select_users_for_email(ids: *sequ(int)*)" and "u:select_users_for_conf(ids: *sequ(int)*)". "ids" specifies the set of all invited users, but not the set of users actually joining the conference. From that initial set of users, the set of actual conference participants is generated by the system actions "s:update_participants(i)" and "s:update_invited(i)" which after each "u:join_conf" or "u:not_join_conf" remove the invited user with the identifier "i" from "ids". The instances "inv_participants" and "conf_participants" are correspondingly updated.

  In the case of email, ids already contains identifiers of all message recipients. Notification on incoming messages with a subject related to a document is specified by the system action "s:rel_message_notif".

The informal specification has been transformed into the following formal specification:

interface loc_istruct: u:open_istruct, u:close_istruct, u:sel_node, u:sel_node_for_editing, u:insert_node, u: del_node, u:move_subtr, u:copy_subtree, u:update_attr, u:intend_to_lock, s:expl_lock, s:display_lock_not_possible, s:impl_lock, s:end_lock, s:int_lock, s:update_version, s:conflict_notif, u:select_users_for_conf, u:select_users_for_email, s:invite_to_conf, u:join_conf, u:not_join_conf, s:rel_message_notif, s:update_participants, s:update_invited, curr_node, global_version, local_version, selected_users, initiator, is_locked, locked_node, version_no, initiator_id, inv_participants, conf_participants;
contains loc_email, loc_text_editor, loc_conf, loc_user_info, loc_host_info, loc_doc_info;
based on *structured_object*, *sequ(nat)*, *sequ(int)*, *bool*;
instances:
global_version, local_version: *structured_object*;

curr_node, locked_node: *sequ(nat)*;
version_no, initiator_id: *int*;
selected_users, inv_participants, conf_participants: *sequ(int)*;
initiator, is_locked: *bool*;

actions:

u:open_istruct(t: *structured_object*), u:close_istruct(), u:sel_node(ns: sequ(nat)), u:sel_node_for_editing(ns: *sequ(nat)*), u:insert_node(node: *struct_obj_node*), u: del_node(n: *nat*), u:move_subtr(n1: *nat*, ns2: *sequ(nat)*, n2: *nat*), u:copy_subtree(n1: *nat*, ns2: *sequ(nat)*, n2: *nat*), u:update_attr(s: *string*), u:intend_to_lock, s:expl_lock(ns: *sequ(nat)*), s:display_lock_not_possible(), s:impl_lock(), s:end_lock(), s:int_lock(), s:update_version(t: *structured_object*), s:conflict_notif(), u:select_users_for_conf(ids: *sequ(int)*), u:select_users_for_email(ids: *sequ(int)*), s:invite_to_conf(i: *int*), u:join_conf(), u:not_join_conf(), s:rel_message_notif(), s:update_participants(i: *int*), s:update_invited(i: *int*);

laws: t: *structured_object*, ns, ns', n, n1, n2: *nat*, ns2: *sequ(nat)*, node: *struct_obj_node*, v, i: *int*, ids: *sequ(int)*, mb: *mbox*, f: *file*, s: *string*;

/* *Preconditions* */
u:intend_to_lock() $\rightarrow$ compare(global_version, local_version);
/* *Postconditions* */
post.u:open_istruct(t) $\rightarrow$ local_version = t;
post.u:select_node(node) $\rightarrow$ curr_node = ns *until* u:select_node(ns');
(u:insert_node(node) $\wedge$ t = local_version) $\rightarrow$ (local_version = insert_struct_obj_node (curr_node, get_num_of_succ(select_node(curr_node,t) + 1), node, t) *atnext post*.u:insert_node (node));
(u:del_node(n) $\wedge$ t = local_version) $\rightarrow$ (local_version = delete_struct_obj_node(curr_node, n, t) *atnext* post.u:del_node(n));
(u:move_subtr(n1,ns2,n2) $\wedge$ t = local_version) $\rightarrow$ (local_version = move_subtree(curr_node, n1, ns2, n2, t) *atnext post*.u:move_subtr(n1, ns2, n2));
(u:copy_subtr(n1,ns2,n2) $\wedge$ t = local_version) $\rightarrow$ (local_version = copy_subtree(curr_node, n1, ns2, n2, t) *atnext post*.u:copy_subtr(n1, ns2, n2));
(u:update_attr(s) $\wedge$ t = local_version $\wedge$ curr_node $\neq$ empty_sequ_nat) $\rightarrow$ (local_version = insert_struct_obj_node(lead(curr_node), last(curr_node), set_string (select_node(curr_node, t), s), delete_struct_obj_node (lead(curr_node), last(curr_node), t)) *atnext post*.u: update_attr(s));
(u:update_attr(s) $\wedge$ t = local_version $\wedge$ curr_node = empty_sequ_nat) $\rightarrow$ (local_version = add_struct_obj_node(set_string(get_node(t), s), empty_struct_obj_node) *atnext post*. u:update_attr(s));
post.s:expl_lock(ns) $\rightarrow$ (is_locked $\wedge$ locked_node = ns) *until* s:end_lock());
post.s:end_lock() $\rightarrow$ $\neg$ is_locked *unless* s:expl_lock(ns);
((*post*.u:insert_node(node) $\vee$ *post*.u:del_node(n) $\vee$ *post*.u:move_subtr(n1, ns2, n2) $\vee$ *post*.u:copy_subtr(n1, ns2, n2) $\vee$ *post*.u:update_attr(s)) $\wedge$ is_subsequ(locked_node, curr_node)) $\rightarrow$ $\bigcirc$ s:end_lock();
loc_text_editor: close() $\rightarrow$ $\bigcirc_{loc\_text\_editor}$ loc_istruct: end_lock();

(s:update_version(t) $\wedge$ v = version_no) $\rightarrow$ ((version_no = v + 1 $\wedge$ global_version = t) *atnext* *post*.s:update_version(t));

(s:update_version(t) $\wedge$ compare(global_version, local_version)) $\rightarrow$ (local_version = t *atnext* *post*.s:update_version(t));

(s:update_version(t) $\wedge$ local_version = t' $\wedge$ $\neg$ compare(global_version, local_version)) $\rightarrow$ (($\bigcirc$ conflict_notif() $\wedge$ t' = local_version) *atnext* *post*.s:update_version(t));

/* *No action may be executed between* u:intend_to_lock() *and* s:expl_lock(ns) *or* s:display_lock_not_possible */

(u:intend_to_lock() $\wedge$ ns = curr_node) $\rightarrow$ $\neg$ (u:insert_node(node) $\vee$ u:del_node(n) $\vee$ u:move_subtr(n1, ns2, n2) $\vee$ u:select_node(ns') $\vee$ u:select_for_editing(ns) $\vee$ u:copy_subtr (n1, ns2, n2) $\vee$ u:update_attr(s) $\vee$ u:intend_to_lock()) *until* s:expl_lock(ns) $\vee$ s:display_lock_not_possible()));

/* *Fairness for getting a lock* */

u:intend_to_lock() $\wedge$ ns = curr_node $\wedge$ t = local_version) $\rightarrow$ $\diamond$ ((get_string(select_node (ns, local_version)) = get_string(select_node(ns, t)) $\wedge$ select_node(ns,local_version) $\neq$ empty_struct_obj_node $\wedge$ expl_lock(ns)) $\vee$ ($\neg$(get_string(select_node (ns, local_version)) = get_string(select_node(ns, t)) $\wedge$ select_node(ns, local_version) $\neq$ empty_struct_obj_node) $\wedge$ s:display_lock_not_possible()));

/* *Local distribution of the new version* */

((*post*.u:move_subtr(n1, ns2, n2) $\vee$ *post*.u:copy_subtr(n1, ns2, n2) $\vee$ *post*.u:insert_node (node) $\vee$ *post*.u:del_node(n) $\vee$ *post*.u:update_attr(s) $\vee$ loc_text_editor[i]: *post*.close()) $\wedge$ v = version_no) $\rightarrow$ $\bigcirc$ (version_no = v + 1 $\wedge$ global_version = local_version);

/* *Invocation of loc_text_editor and feedback* */

post.u:select_for_editing(ns) $\wedge$ t = local_version $\wedge$ ns $\neq$ empty_sequ_nat) $\rightarrow$ $\bigcirc$ loc_text_editor: (open(get_document(select_node(ns, t))) $\rightarrow$ $\Box_{loc\_text\_editor}$ loc_text_editor: ((*post*.save() $\wedge$ f = act_file) $\rightarrow$ $\bigcirc_{loc\_text\_editor}$ loc_istruct: local_version = insert_struct_obj_node(lead(ns), last(ns), make_document_node (select_node(ns,t), f, delete_struct_obj_node(lead(ns), last(ns), t)))));

post.u:select_for_editing(ns) $\wedge$ t = local_version $\wedge$ ns = empty_sequ_nat) $\rightarrow$ $\bigcirc$ loc_text_editor: (open(get_document(select_node(ns, t))) $\rightarrow$ $\Box_{loc\_text\_editor}$ loc_text_editor: ((*post*.save() $\wedge$ f = act_file) $\rightarrow$ $\bigcirc_{loc\_text\_editor}$ loc_istruct: local_version = add_struct_obj_node (make_document_node (select_node(ns,t), f, empty_struct_obj)));

/* *Conference activation* */

/* *Postconditions* */

post.s:invite_to_conf(i) $\rightarrow$ (i = initiator_id $\wedge$ $\diamond$ (u:join_conf() $\vee$ u: not_join()));

post.u:select_users_for_conf(ids) $\rightarrow$ initiator *unless* s:invite_to_conf(i);

post.s:invite_to_conf() $\rightarrow$ ($\neg$ initiator *unless* u:select_users_for_conf(ids));

post.u:select_users_for_conf(ids) $\rightarrow$ inv_participants = ids $\wedge$ conf_participants = empty_sequ_int;

(s:update_participants(i) $\wedge$ ids = conf_participants $\wedge$ ids' = inv_participants) $\rightarrow$ ((conf_participants = insert_sequ(ids, i) $\wedge$ inv_participants = delete_sequ_int(ids',i))) *atnext* *post*.s:update_participants(i));

(s:update_invited(i) $\wedge$ ids' = inv_participants) $\rightarrow$ (inv_participants = delete_sequ_int (ids',i)) *atnext* *post*.s:update_invited(i));

/* *Interaction with email* */

(*post*.u:select_users_for_email(ids) $\wedge$ s = get_string(get_node(local_version))) $\rightarrow$ $\bigcirc$ loc_email: compose_message(s, ids);

loc_email: (mb = mbox $\rightarrow$ $\bigcirc$ ((num_of_messages_in_mbox(mb) = num_of_messages_in_mbox (mailbox)) = s) $\rightarrow$ $\bigcirc$ loc_istruct: (s = get_string(get_node(local_version)) $\rightarrow$ s:rel_message_notif()));

/* *Interaction with awareness tools* */

post.u:open_istruct(t) $\rightarrow$ $\bigcirc$ (loc_user_info: s:open_user_info() $\wedge$ loc_host_info: s:open_host_info() $\wedge$ loc_doc_info: s:open_doc_info());

post.u:close_istruct(t) $\rightarrow$ $\bigcirc$ (loc_user_info: s:close_user_info() $\wedge$ loc_host_info: s:close_host_info() $\wedge$ loc_doc_info: s:close_doc_info());

init:

global_version = empty_structured_object; local_version = empty_structured_object;
curr_node = empty_sequ_nat; locked_node = empty_sequ_nat;
version_no = 0; initiator_id = -1;
selected_users = empty_sequ_int;
inv_participants = empty_sequ_int;
conf_participants = empty_sequ_int;
$\neg$ initiator; $\neg$ is_locked;

endinterface loc_istruct;

/* *Part of the system-interface related to loc_istruct* */

system-interface: t: *structured_object*, ns, ns', n, n1, n2: *nat*, ns2: *sequ(nat)*, node: *struct_obj_node*, s: *string*, v, i, j: *int*, ids': *sequ(int)*, mb: *mbox*;

/* *Mutual exclusion defined by a lock* */

$\bigwedge_{1\leq i\leq n} \bigwedge_{1\leq j\leq n, i\neq j}$ (($\Diamond_{loc\_istruct[i]}$ loc_istruct[j]: s:expl_lock(ns) $\wedge$ $\Diamond_{loc\_istruct[i]}$ loc_istruct[i]: (((u:move_subtr(n1, ns2, n2) $\vee$ u:copy_subtr(n1, ns2, n2) $\vee$ u:insert_node(node) $\vee$ u:del_node(n) $\vee$ u:update_attr(s)) $\wedge$ is_subsequ(ns, curr_node)) $\vee$ (u:select_for_editing(ns') $\wedge$ is_subsequ(ns, ns')))) $\rightarrow$ ((loc_istruct[j]: *post*.s:end_lock() $before_{loc\_istruct[i]}$ loc_istruct[i]: (((u:move_subtr(n1, ns2, n2) $\vee$ u:copy_subtr(n1, ns2, n2) $\vee$ u:insert_node(node) $\vee$ u:del_node(n)) $\vee$ u:update_attr(s)) $\wedge$ is_subsequ(ns, curr_node)) $\vee$ (u:select_for_editing(ns') $\wedge$ is_subsequ(ns, ns'))) $\vee$ (loc_istruct[i]: (((*post*.u:move_subtr(n1, ns2, n2) $\vee$ *post*.u:copy_subtr(n1, ns2, n2) $\vee$ *post*.u:insert_node(node) $\vee$ *post*.u:del_node(n) $\vee$ *post*.u:update_attr (s)) $\wedge$ is_subsequ(ns, curr_node)) $\vee$ loc_text_editor[i]: *post*.close()) $before_{loc\_istruct[i]}$ loc_istruct[j]: s:expl_lock(ns))));

/* *Distribution of the new version* */

loc_struct[i]: ((*post*.u:move_subtr(n1, ns2, n2) $\vee$ *post*.u:copy_subtr(n1, ns2, n2) $\vee$ *post*.u:insert_node(node) $\vee$ *post*.u:del_node(n) $\vee$ *post*.u:update_attr(s) $\vee$ loc_text_editor[i]: *post*.close()) $\wedge$ t = local_version) $\rightarrow$ $\bigwedge_{1\leq j\leq n, i\neq j}$ $\bigcirc_{loc\_istruct[i]}$ loc_istruct[j]: update_version(t);

/* *Activation of a conference* */

loc_istruct[i]: *post*.u:select_users_for_conf(ids) $\rightarrow \bigwedge_{1 \leq j \leq n, is\_in\_sequ(ids,j)} \bigcirc_{loc\_istruct[i]}$ loc-_istruct[j]: s: invite_to_conf(i);
loc_istruct[i]: (*post*.u:join_conf() $\wedge$ j = initiator_id) $\rightarrow \bigcirc_{loc\_istruct[i]}$ loc_istruct[j]: update_participants(i);
loc_istruct[i]: (*post*.u:not_join_conf() $\wedge$ j = initiator_id) $\rightarrow \bigcirc_{loc\_istruct[i]}$ loc_istruct[j]: update_invited(i);
loc_istruct[i]: ((*post*.s:update_participants(i) $\vee$ *post*.s:update_invited(i)) $\wedge$ inv_participants = empty_sequ_int $\wedge$ conf_participants $\neq$ empty_sequ_int $\wedge$ ids' = conf_participants) $\rightarrow \bigwedge_{1 \leq j \leq n, is\_in\_sequ(ids',j)} \bigcirc_{loc\_istruct[i]}$ loc_conf[j]: s:open_conf();
loc_istruct[i]: (*post*.u:take_audio() $\rightarrow \bigwedge_{1 \leq j \leq n, i \neq j} \bigcirc_{loc\_istruct[i]}$ loc_conf[j]: active_audio $\rightarrow$ s:release_audio());

Some laws of the system-interface of istruct related to the audio-video conference interface have been already defined in loc_conference.

3.2.6 Specification of the system-interface

The parts of the system-interface related to different applications have already been specified in the previous subsections. The complete system-interface is therefore defined as follows:

system Iris:
consists of loc_istruct[1], ..., loc_istruct[n];
based on
/* *all sorts of the variables involved in the laws of the system-interface specifications for email, audio-video conference, awareness tools and structure editor* */
laws:
/* *laws as defined in the system-interface specifications for email, audio-video conference, awareness tools and structure editor* */
endsystem Iris;

Considering the contents of individual local interface specifications and the system-interface specification, the initially emphasized aspects of user interaction and cooperation may be observed:

- Actions are specified as user and system actions depending on who initiates them. Only externally observable system actions of interfaces are specified.
- Asynchronous communication has been specified in the email interface and synchronous communication in the audio-video conference interface. Including specifications of both tools into the istruct interface specification, their combination can be observed.

- In the specification of the audio-video conference interface, audio and video are described by different sorts. Other media involved in the specification are text and graphichs.

- Cooperation between users can be observed in the specifications of all interfaces which contain the system-interface part. Cooperation awareness is described by specification of awareness tools.

- Involvement of a social protocol has been observed in the audio-video conference interface and in the istruct interface. Despite that the intermediate document versions are distributed to all participants after each structure operation and are treated as temporarily global, the final global version of a document may be defined only upon agreement of all participants working on a document.

For better presentation of the system-interface specification of Iris, we give in table 3.2 an overview of all its building blocks. Definition of all operations, actions, parameters and instances involved can be found in the previous subsections.

specification components		**total**
system		
Iris		1
compound interfaces (per user)		
loc_istruct loc_email		2
interfaces (per user)		
loc_text_editor loc_editor_in_email loc_conf loc_user_info loc_host_info loc_doc_info		6
sorts		
generally used: *int, sequ(int), nat, sequ(nat), string, string_sequ, bool, file*	8	
used in loc_text_editor: *editor*	1	
used in loc_conf: *audio, protocol_message, graphic, window,*		
window_sequ	5	
used in the interfaces of awareness tools: *user, subset_of_users, set_of_users, host, set_of_hosts, replica, version, access_right, doc_info, set_of_replicas, set_of_curr_vers, set_of_access_rights, set_of_documents*	13	
used in loc_istruct: *document, struct_obj_node, structured_object*	3	
used in loc_email: *message, mbox*	2	32

Table 3.2: Specification components of the system Iris

Chapter 4

Use of the method

4.1 Methodological guidelines

The proposed formal method can not efficiently be used without definition of an accompanied specification methodology, because only a formal method cannot provide a systematical translation of an informal description into a formal one. The methodological guidelines described in the following are based on our experience collected by using the method for specification of different examples. Our experience also shows that a methodology can only be precisely defined on the basis of complex examples of real applications, in our case Iris.

The input of our method is an informal requirements specification or a detailed informal description of an existing application. Its output is a complete formal specification of the functionality required at the input. The method can be used for specification of an application constructed from scratch or from existing components.

A CADT specification is constructed using the following steps (the methodology outline is shown in figure 4.1):

- First, a *detailed analysis* of the given informal requirements for a new CSCW application or the description of an existing application is done. For the case of user interfaces, a precise description of this activity can be found in [Jac97].

- On the basis of the structure of component calls defined or recognized in the first step, the *architecture of interfaces* is defined.

  For each interface, a *precise informal specification* is written, since the existing application decriptions, manuals and documentation usually do not have the appropriate form and contents to be directly used for translation into a complete formal specification. That was needed also in the example of Iris (see for example the informal specification of the istruct functionality in subsection 3.2.5 which we wrote on the basis of the existing descriptions in [Koch95a, Koch95b, Iris96, Koch97]).

- If the new or existing CSCW application supports cooperation between users controlled by *rules of a social protocol*, a formal or semiformal specification of those

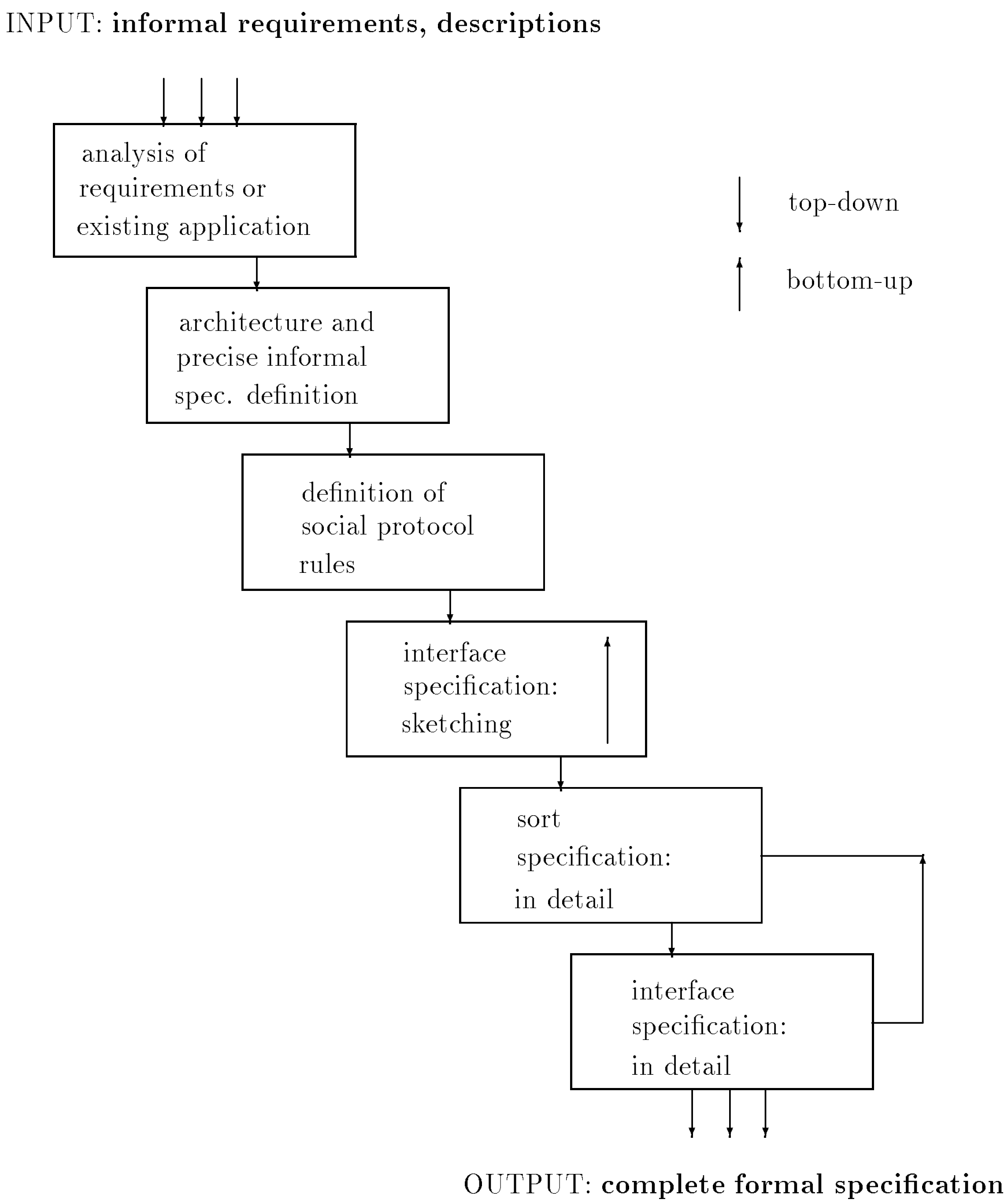

Figure 4.1: Methodology outline

rules is created. As already mentioned, the specification of social protocol rules should not be a part of the application specification, but its contents has to be known while specifying the application.

If a CSCW application is being constructed from existing and already specified components which also involve some kind of a social protocol, social protocol rules of components have to be taken into account when the social protocol rules of the new application are specified. We suggest to use formal or semiformal specifications of social protocol rules to decrease ambiguity as much as possible.

- *Sketches of interface specifications* are made where the the actions, sorts and operations needed are identified. The partition of the functionality to be represented either by actions of interfaces or by operations of sorts completely depends on the concrete example of application. Only a general advice can be given that the functionality depending very much on the contents of data structures could be specified by operations of sorts in the most convenient way.

- *Sorts* are specified in detail.

- *Interfaces* are specified in detail. If detected that the defined operations in sorts are not sufficient for the specification of the interfaces, the operation definitions have to be changed or additional operations have to be specified. The last two steps are repeated until the application is completely specified.

Generally, we use the top-down approach to create CADT specifications which is partly combined with the bottom-up approach. The combination is needed:

- in the step creating sketches of interface specifications, where lower-level local interfaces are described before higher-level local interfaces and the system-interface,

- between the steps of the sort definition and the detailed interface definition as described above.

4.2 Tool support

A formal approach or method can successfully be transferred from the academic research to practical use only if it is sufficiently supported by software tools. This is particularly true in the case of our method, since abstract data types and temporal logic specifications are not easy to create and understand by users who have no experience with that kind of formalisms. To bring our method closer to practical use, the following tasks in different development phases of a CSCW application should be supported by tools:

- First, an *editor* is needed which would make the task of writing a specification much easier. An appropriate solution would probably be a graphic editor where graphic symbols for different building blocks and logic operators would be available. The editor should also provide facilities for defining schemes (as described in the next section) and syntax checking. A user should have the possibility to choose between two output forms of the created specification: a graphical and a textual form (as available, for example, in the VERILOG OG Object Editor for the specification language SDL [Ver97]).

Using LaTeX as in the example of this report, an output in the paper or postcript form of high quality can be produced, but the specification contained in the source ASCII file is, due to many LaTeX commands, time-consuming to write, not easy to read and hardly usable as an input to other tools (for syntax or model checking, for example). For a simple syntax checking only, the GNU Emacs could be adapted.

- If a CSCW application is being constructed from existing (and already specified) components, it should be checked if a newly added component together with other components in the specification satisfies the required behaviour. This task can considerably be simplified by using a *theorem prover*. There are several logical frameworks with theorem-proving capabilities already available as, for example, Larch prover, Isabelle, HOL prover or Lambda theorem prover. They cannot be applied on a temporal logic specification written in DistTL without further research how the object logic (DistTL) can be represented in a meta-logic provided by a standard logical framework. Much work on representations (i.e. embeddings) in different logical frameworks has been done for TLA (see, for example, [Lang94, Kal95, Busch95, Merz95]).

- After defining a formal model of an implementation of a CSCW application, a *model checker* would be needed to check if the implementation satisfies the requirements of the specification. Not much research has been done yet to define a formal model of the implementation, but we estimate that Petri nets might be very useful to model concurrency and verify the implementation properties.

4.3 Improving clarity of specifications

The clarity of specifications could be significantly improved by extension of CADT by *facilities for defining schemes.* In the example of Iris specification we identified some typical situations, where schemes could be used:

- *Definition of sorts.* Many sorts have very similar operations, as for example the sorts *message*, *mbox*, *protocol message*, *user*, *doc_info* etc.

  They are based on different sorts, but all have similar operations, typically "insert", "get" and "remove" related to different parameters. A scheme for a sort definition could be used where from a generalized sort containing those operations its specialization with concrete names and parameters is to be generated. A user would only enter the name of the new sort, the list of sorts based on, and the lists of parameters and their types, but no laws which are assumed to be added automatically. "sort" may represent a parameter type of any sort.

define sort: *sort name*
based on *list of sorts*
operations
  insert *(par1:sort,...,parN:sort):sort*;
  ... /* other insert operations */
  get *(par1:sort,...,parN:sort):sort*;
  ... /* other get operations */
  remove *(par1:sort,...,parN:sort):sort*;
  ... /* other remove operations */

- *Mutual exclusion.* On the level of the system-interface, mutual exclusion is often required between actions belonging to different local interfaces. For the sections from the action *act1* to *act2* of the interface *i1* and from the action *act3* to *act4* of the interface *i2*, mutual exclusion can be specified using the scheme:

  define mutual exclusion between: *(i1:act1()* **to** *act2())* **and** *(i2:act3()* **to** *act4())*

- *Only one instance of an interface active at the same time.* That requirement is usually specified by the laws (in the interface loc_host_info, for example):

  opening action $\rightarrow \neg$ *opening action until closing action*;
  start $\vee$ *closing action* $\rightarrow$ (*opening action before closing action* $\vee$ other actions);

  It could be replaced by the scheme:

  define one instance active: *(opening action, closing action)* **other actions** (*list of other actions)*

- *Intialization of instances.* Instances used in a local interface are typically set to an initial value by using the constant "zero" operation of the referred sort. The scheme

  define init: *(instance name 1: sort, ..., instance name N: sort)*

  would allow a user only to enter the instance names and the related sorts. The approapriate constant operations could be added automatically.

Introduction of schemes requires further theoretical work on formal definition of schemes in CADT and identification of more situations where a scheme can hide details of some specification parts from a user. Those situations can be identified only by a detailed analysis of specifications created for real applications. As already mentioned, introduction of schemes requires also an appropriate tool support.

Chapter 5

Conclusion

5.1 Results

We have suggested a formal method for specification of CSCW applications and shown its use for specification of multi-user editor Iris. We specified the interface layer, some elements of the access layer for its central part - structure editor - and some external components. It has also been shown how a social protocol supported in a CSCW application can be considered in a formal specification. Methodological guidelines for use of the method have been given and software tools have been identified required to support the use. Except the disadvantage that interfaces cannot be created dynamically, the proposed combination of CADT (Concurrent Abstract Data Types) and DistTL (Distributed Temporal Logic) has been proved as appropriate for specification of Iris.

5.2 Future work

We intend to continue the work described in this report in the following directions:

- The method should be extended to provide the possibility of dynamic creation of interfaces. As a promising solution, we see its integration with an object oriented model.

- Another extension is required to be able to specify real-time characteristics that is particularly important for specification of multimedia applications. The extension might be done by introduction of timed operations as, for example, in metric temporal logic (MTL) [Koy90] or quality of service temporal logic (QTL) [Bow94], or explicit timer variables as in real-time temporal logic (RTTL) [Ost90] or TLA [Lam94].

- Further theoretical work has to be done to improve clarity of specifications by introducing schemes. CADT have to be extended with a formal definition of schemes. In addition, transformations and corresponding algorithms have to be developed to generate a CADT specification in the usual extended form (defined in chapter 2) from a CADT specification containing schemes.

- Further study of cooperation based on a social protocol should be done in order to determine how precisely rules of a social protocol can formally be specified and to what extent a social protocol can affect a formal specification of a CSCW application.

- As already described in the previous chapter, software tools should be developed to support the use of the method.

Acknowledgements

Most of the presented research was done during my stay at Lehrstuhl Angewandte Informatik, verteilte Systeme, Institut für Informatik, Technische Universität München, as a visiting researcher under DAAD grant. I would like to thank Prof. Dr. Johann Schlichter and Martina Nöhmeier (Lehrstuhl Angewandte Informatik, verteilte Systeme), and Maximilian Frey and Dr. Holger Schlingloff (Lehrstuhl Programmierung, verteilte Programmierung) for constructive discussions and comments on relevance of the suggested method for CSCW applications or on its theoretical foundations.

Bibliography

[Aalst94] M.P. van de Aalst, K.M. van Hee, G.J. Houben, Modelling workflow management with high level Petri nets, *Proc. Workshop On Computer Supported Cooperative Work, Petri Nets and Related Formalisms*, Zaragoza, Spain, 1994, pp. 31 – 50.

[Ago94] A. Agostini, G. de Michelis, K. Petruni, Keeping workflow models as simple as possible, *Proc. Workshop On Computer Supported Cooperative Work, Petri Nets and related formalisms*, Zaragoza, Spain, 1994, pp. 11 – 29.

[Benf93] S. Benford, J. Palme, A standard for OSI group communication, *Computer Networks and ISDN Systems*, Vol. 25, 1993, pp. 933 – 946.

[Berry96] A. Berry, *A Panacea for Distributed CSCW Infrastructure?*, Dissertation (Abstract), University of Queensland, 1996.

[Bing97] M. Bing, X. Li, An object-based model for prototyping user interfaces of cooperative systems, *Software Engineering Notes*, Vol. 22, No. 2, 1997, pp. 72 – 76.

[Bland97] A.E. Blandford, D.J. Duke, Integrating user and computer system concerns in the design of interactive systems, *Int. J. Human-Computer Studies*, Vol. 46, No. 5, 1997, pp. 653 – 680.

[Bov90] S. Bovair, D.E. Kieras, P. Polson, The aquisition and performance of text-editing skill: a cognitive complexity analysis, *Human Computer Interaction*, Vol. 5, 1990, pp. 1 – 48.

[Bow94] H. Bowman, G. Blair, L. Blair, A. Chetwynd, Time versus abstraction in formal descriptions, R.L. Tenney, P.D. Amer, and M.Ü. Uyar, Eds., *Formal Description Techniques VI*, Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1994, pp. 467 – 482.

[Brau94] T. Braun, M. Zitterbart, Videokonferenzen im Internet, *Praxis der Informationsverarbeitung und Kommunikation*, Vol. 17, No. 4, 1994, pp. 253 – 254.

[Busch95] H. Busch, A practical method for reasoning about distributed systems in a theorem prover, E.T. Schubert, P.J. Windley, J. Alves-Foss, Eds., *Workshop on*

Higher-Order Logic Theorem Proving and Its Applications, *LNCS 971*, Springer-Verlag, Berlin, 1995, pp. 106 – 121.

[Chi85] U.H. Chi, Formal specification of user interfaces: A comparison and evaluation of four axiomatic approaches, *IEEE Trans. Software Engineering*, Vol. SE-11, 1985, pp. 671 – 685.

[Dew93] P. Dewan, Tools for implementing multiuser user interfaces, *Trends in Software*, Vol. 1, 1993, pp. 149 – 172.

[Dew96] P. Dewan, Multiuser architectures, *Proc. of IFIP WG2.7 Working Conference on Engineering for Human-Computer Communication*, 1995, Chapman & Hall, London, 1996, pp. 247 – 270.

[Diaz96] M. Diaz, T. Villemur, F. Vernadat, Formal design of cooperative systems, P. Dembiński, M. Średniawa, Eds., *Protocol Specification, Testing and Verification XV*, Chapman & Hall, London, 1996, pp. 349 – 365.

[Dol91] J. Dollimore, S. Wilbur, Experiences in building a configurable CSCW system, J.M. Bowers and S.D. Benford, Eds., *Studies in Computer Supported Cooperative Work*, Elsevier, Amsterdam, 1991, pp. 173 – 181.

[Dou96] P. Dourish, *Open Implementation and Flexibility in CSCW Toolkits*, Dissertation, University College London, 1996.

[Ehr85] H. Ehrig, B. Mahr, *Fundamentals of Algebraic Specification 1*, Springer Verlag, Berlin Heidelberg, 1985.

[Elli91] C.A. Ellis, S.J. Gibbs, G.L. Rein, Groupware: Some issues and experiences, *Communications of the ACM*, Vol. 34, 1991, pp. 38 – 58.

[Emer90] E.A. Emerson, Temporal and modal logic, J. van Leeuwen, Ed., *Handbook of Theoretical Computer Science*, Vol. B, Elsevier, Amsterdam, 1990, pp. 996 – 1072.

[Fisch96a] S. Fischer, Implementation of multimedia systems based on a real-time extension of Estelle, R. Gotzhein and J. Bredereke, Eds., *Formal Description Techniques IX: Theory, application and tools*, Chapman & Hall, London, 1996, pp. 310 – 326.

[Fisch96b] S. Fischer, On the suitability of Estelle for multimedia systems, P. Dembiński, M. Średniawa, Eds., *Protocol Specification, Testing and Verification XV*, Chapman & Hall, London, 1996, pp. 369 – 384.

[Frey97] M. Frey, M. Pučko, Formal specification of CSCW applications with concurrent abstract data types, to appear in *Journal of Systems Architecture*, Elsevier Science, 1997.

[Grah92] T.C. Nicholas Graham, T. Urnes, Relational views as a model for automatic distributed implementation of multi-user applications, *Proc. of the CSCW 92 Conf.*, ACM, 1992, pp. 59 – 66.

[Grah97] T.C. Nicholas Graham, T. Urnes, Integrating support for temporal media into an architecture for graphical user interfaces, *Proc. of the ICSE 97 Conf.*, Boston, MA, 1997, ACM, pp. 172 – 182.

[Hart90] R. Hartson, A.C. Siochi, D. Hix, The UAN: A user-oriented representation for direct manipulation interface designs, *ACM Transactions on Information Systems*, Vol. 8, 1990, pp. 181 – 203.

[Huhn96] M. Huhn, P. Niebert, F. Wallner, *Put Your Model Checker on Diet: Verification on Local States*, Technical Report TUM-I9642, Inst. für Informatik, Techn. Univ. München, 1996.

[Iris96] Iris Online Manual, http://www11.informatik.tu-muenchen/software/iris/

[ISO83] ISO – Information Processing Systems – "Basic Reference Model for Open Systems Interconnection", *IS 7498*, 1983.

[ISO89] ISO – Information Processing Systems – Open Systems Interconnection – "LOTOS - A formal description technique based on temporal ordering of observational behaviour", *IS 8807*, 1989.

[ITU-T93] ITU-T, *Recommendation Z.100*, CCITT Specification and Description Language (SDL), 1993.

[ITU-T94] ITU-T, *Recommendation Z.120*, Message Sequence Chart (MSC), 1994.

[Jac97] J.-P. Jacquot, D. Quesnot, Early specification of user-interfaces: toward a formal approach, *Proc. of the ICSE 97 Conf.*, Boston, MA, 1997, ACM, pp. 150 – 160.

[John91] C.W. Johnson, D. Diaper, N. Hammond, Applying temporal logic to support the specification and prototyping of concurrent multi-user interfaces, *Proc. of the HCI 91 Conference*, Cambridge Univ. Press, 1991, pp. 145 – 156.

[Jon90] D. Jones, *Systematic Software Development Using VDM*, Prentice-Hall, 2nd edition, 1990.

[Kal95] S. Kalvala, A formulation of TLA in Isabelle, http://www.cl.cam.ac.uk/users/sk/, 1995.

[Kerb95] A. Kerbrat, S.B. Atallah, Formal specification of a framework for groupware development, G. v. Bochmann, R. Dssouli and O. Rafiq, Eds., *Formal Description Techniques VIII*, Chapman & Hall, London, 1995, pp. 303 – 310.

[Kleu95] S. Kleuker, A. Kehne, H. Tjabben, *Provably Correct Communication Networks (CoCoN)*, Philips Research Laboratories Aachen, Technical Report, 1123/95, 1995.

[Kleu96a] S. Kleuker, Using formal methods in the development of protocols for multi-user multimedia systems, R. Gotzhein and J. Bredereke, Eds., *Formal Description Techniques IX: Theory, application and tools*, Chapman & Hall, London, 1996, pp. 113 – 128.

[Kleu96b] S. Kleuker, H. Tjabben, *A Formal Approach to the Development of Reliable Multi-User Multimedia Applications*, Philips Research Laboratories Aachen, Technical Report, 1168/96, 1996.

[Knir95] P. Knirsch, *Kaskadenrekursion in verteilten Systemen*, Dissertation, Inst. für Informatik, Tech. Univ. München, 1995.

[Koch95a] M. Koch, Design issues and model for a distributed multi-user editor, *Computer Supported Cooperative Work*, Vol. 3, No. 4, 1995, pp. 359 – 378.

[Koch95b] M. Koch, *The Collaborative Multi-User Editor Project IRIS*, Technical Report TUM-I9524, Inst. für Informatik, Techn. Univ. München, 1995.

[Koch97] M. Koch, *Unterstützung kooperativer Dokumentenbearbeitung in Weitverkehrsnetzen*, Dissertation, Inst. für Informatik, Tech. Univ. München, 1997.

[Koy90] R. Koymans, Specifying real-time properties with metric temporal logic, *Real-Time Systems Journal*, Vol. 2, No. 4, 1990, pp. 255 – 299.

[Kroe87a] F. Kröger, Abstract modules: Combining algebraic and temporal logic specification means, *Technique et Science Informatiques*, Vol. 6, 1987, pp. 559 – 573.

[Kroe87b] F. Kröger, *Temporal Logic of Programs*, Springer Verlag, Berlin, 1987.

[Lang94] T. Långbacka, A HOL formalisation of the Temporal Logic of Actions, T.E. Melham, J. Camilleri, Eds., *High Order Logic Theorem Proving and Its Applications*, *LNCS 859*, Springer-Verlag, Berlin, 1994, pp. 332 – 345.

[Lam94] L. Lamport, The temporal logic of actions, *ACM TOPLAS*, Vol. 16, No. 3, 1994, pp. 872 – 943.

[Less95] F. Leßke, *Abstrakte Datentypen und temporale Logik: Ein kombinierter Spezifikationsansatz*, Dissertation, Fakultät für Informatik, Ludwig – Maximilians – Universität München, 1995.

[Lod87] K. Lodaya, P.S. Thiagarajan, A modal logic for a subclass of event structures, *Proc. 14. Int. Colloquium on Automata, Languages and Programming ICALP '87*, Karlsruhe, 1987, *LCNS 267*, Springer Verlag, 1987, pp. 290 – 303.

[Lod92] K. Lodaya, R. Ramanujam, P.S. Thiagarajan, Temporal logics for communicating sequential agents: I, *Int. Journal of Foundations of Computer Science*, Vol. 3, No. 2, 1992, pp. 117 – 159.

[MBo96] The MBone Information Web, *vic – Video Conferencing Tool*, http://www-nrg.ee.lbl.gov/vic/, 1996.

[Merz95] S. Merz, Mechanizing TLA in Isabelle, *Proc. Workshop Verification in New Orientations*, Maribor, Slovenia, 1995, Technical Report 03/95, University of Maribor.

[Oppen89] A.V. Oppenheim, R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall Signal Processing Series, Prentice-Hall, Englewood Cliffs, N.J., 1989.

[Ost90] J.S. Ostroff, *Temporal Logic of Real-Time Systems*, Research Studies Press, 1990.

[Payne84] S.J. Payne, Task-action grammars, B. Shackel, Ed., *Human-Computer Interaction: Interact '84*, Elsevier, Amsterdam, 1984.

[Poit95] S. Poitrenaud, The PROCOPE semantic network: an alternative to action grammars, *Int. J. Human-Computer Studies*, Vol. 42, No. 1, 1995, pp. 31 – 70.

[Puer97] A.R. Puerta, A model-based interface development environment, *IEEE Software*, Vol. 14, No. 4, 1997, pp. 40 – 47.

[Rei88] W. Reisig, Temporal logic and causality in concurrent systems, Concurrency 88, *Lecture Notes in Computer Science*, Vol. 335, Springer Verlag, Berlin, 1988, pp. 121 – 139.

[Reis81] P. Reisner, Formal grammar and human factors design of an interactive graphics system, *IEEE Trans. Software Engineering*, Vol. SE-3, 1981, pp. 218 – 229.

[Rek96] J. Rekers, I. Sprinkhuizen-Kuyper, A LOTOS specification of a CSCW tool, D. Shapiro, M. Tauber, R. Traunmüller, Eds., *The Design of Computer Supported Cooperative Work and Groupware Systems*, Elsevier Science B.V., 1996, pp. 259 – 277.

[Shen92] H.H. Shen, P. Dewan, Access control for collaborative environments, *Proc. of the CSCW 92 Conference*, ACM, 1992, pp. 51 - 58.

[Smi93] G. Smith, T. Rodden, Access as a means of configuring cooperative interfaces, *Proc. COOCS 93 Conference*, ACM, 1993, pp. 289 – 298.

[Smi95] G. Smith, T. Rodden, SOL: A shared object toolkit for cooperative interfaces, *Int. J. Human-Computer Studies*, Vol. 42, No. 2, 1995, pp. 207 – 234.

[Spiv89] J.M. Spivey, *The Z Notation*, Prentice Hall International Series in Computer Science, 1989.

[Tayl88] M.M. Taylor, Layered protocols for computer-human dialogue I: principles, *Int. Journal of Man-Machine Studies*, Vol. 28, pp. 175 – 218.

[Teeg93] G. Teege, U.M. Borghoff, Combining asynchronous and synchronous collaborative systems, *Proc. 5th Int. Conf. on Human-Computer Interaction*, 1993.

[Ver97] VERILOG, *ObjectGEODE Documentation*, 1997, Tutorial available at http://www.verilogusa.com/.

[Weg95] B. Wegener, *Konzeption und Realisierung eines Videokonferenzmoduls*, Diplomarbeit, Inst. für Informatik, Tech. Univ. München, 1995.

[Whit90] G.M. White, A formal method for specifying temporal properties of the multiuser interface, *Proc. of the IFIP WG 8.4 Conference on Multi-User Interfaces and Applications*, Elsevier, Amsterdam, 1990, pp. 49 – 59.

[Will97] M.H. Willebeek-LeMair, Z.-Y. Shae, Distributed video conferencing systems, *Computer Communications*, Vol. 20, No. 3, 1997, pp. 157 – 168.

[Wirs90] M. Wirsing, Algebraic Specification, J. van Leeuwen, Ed., *Handbook of Theoretical Computer Science*, Vol. B, Elsevier, Amsterdam, 1990, pp. 675 – 788.