COMIC

ESPRIT BASIC RESEARCH PROJECT 6225

A Notation for Computational Mechanisms of Interaction

Abstract

The present report documents the research activities undertaken in Task 3.2 of the COMIC project.

The objective of the three years of research of Strand 3 is to develop a conceptual foundation for designing computational mechanisms of interaction for CSCW applications that can support the complex task of articulating distributed cooperative activities.

The present deliverable argues that a common computational notation for constructing different types of computational mechanisms of interaction is feasible and specifies a three level architecture by means of which the notation in principle makes it possible to construct mechanisms of interaction that are malleable to any degree deemed appropriate for any particular setting.

Document ID	COMIC-D3.3
Status	Final
Type	Deliverable
Version	1.3
Date	September, 1994
Editors	C. Simone (Milano), K. Schmidt (Risø)
Task	3.2

Project coordinator:

Tom Rodden
Computing Department
University of Lancaster
Lancaster LA1 4YR
United Kingdom
Phone: (+44) 1524 593 823
Fax: (+44) 1524 593 608
Email: tom@comp.lancs.ac.uk

The COMIC project comprises the following institutions:

Gesellschaft für Mathematik und Datenverarbeitung (GMD), Bonn, Germany
Risø National Laboratory, Roskilde, Denmark
Royal Institute of Technology (KTH), Stockholm, Sweden
Swedish Institute for Computer Science (SICS), Stockholm, Sweden
Universitat Politècnica de Catalunya (UPC), Barcelona, Spain
University of Amsterdam, Amsterdam, The Netherlands
University of Lancaster, Lancaster, United Kingdom (Coordinating Partner)
University of Manchester, Manchester, United Kingdom
University of Milano, Milano, Italy
University of Nottingham, Nottingham, United Kingdom
University of Oulu, Oulu, Finland

Editors of this report:

Carla Simone
Dept. of Information Sciences
University of Milano
I-20135 Milano
Italy
Phone: (+39) 2 55006-289
Fax: (+39) 2 55006-276
Email: simone@hermes.mc.dsi.unimi.it

Kjeld Schmidt
Cognitive Systems Group
Risoe National Laboratory
P.O. Box 49
DK-4000 Roskilde, Denmark
Phone: (+45) 4677 5146
Fax: (+45) 4675 5170
Email: kschmidt@risoe.dk

ISBN 0 901800 56 2
Lancaster University, 1994
This report is available via anonymous FTP from ftp.comp.lancs.ac.uk.
Further information on COMIC is available from http://www.lancs.ac.uk/computing/research/cseg/comic/

Table of Contents

Introduction

The present report documents the research activities conducted within Task 3.2 of the COMIC project.

The objective of the three years of research of Strand 3 is to develop a conceptual foundation for designing computational mechanisms of interaction that are embedded in CSCW applications in order to support the articulation of cooperative work. Or in the words of the Technical Annex of the COMIC project:

> "The overriding objective of this workpackage is to achieve a clear understanding of the role of mechanisms of interaction in cooperative work and the requirements they must meet in terms of visibility, flexibility, etc. so as to determine the role and requirements of computational notations as means for incorporating mechanisms of interaction in CSCW applications. Based on the results of that research, computational notations for incorporating mechanisms of interaction in CSCW applications will be developed and tried out experimentally." (COMIC, 1992, p. 41).

Within this general plan, the objectives of Task 3.2 have been:

> "(1) to examine existing computational notations with a view to their suitability and applicability for the incorporation of mechanisms of interaction in CSCW applications and (2) to explore the feasibility of a common computational notation suitable for different types of mechanisms of interaction. This task will require an input from social science to inform the evaluation of existing notations in terms of the analysis of Task 3.1." (COMIC, 1992, p. 44).

More specifically, the objective of the research work reported in this deliverable has been to transform the framework for conceptualizing the requirements of mechanisms of interaction, that was developed during the first year's research, into a more systematic construction that should serve as the basis for the implementation activities to be undertaken during the last year of the project.

This construction focuses on two complementary aspects: first of all, a set of requirements of mechanisms of interaction whose definition allows for the identification of specific characteristics of the notation for controlling and specifying the mechanisms; secondly, an initial definition of the notation that embodies these characteristics into a set of syntactic features whose semantics guarantees that the designed mechanisms meet the desired requirements.

The basic choice of keeping these two aspects as separated although coordinated research tracks allowed us to develop the requirements definition without being biased by the possible restrictions imposed by the notation and, at the same time, to develop the notation by focusing on the capabilities provided by current research on the modeling of distributed systems and thus, eventually, make the requirements themselves richer or more precise.

The coordination of these two activities was based on the results provided by the concomitant field studies aimed at illuminating how mechanisms of interaction are defined and exploited in real work settings, in order to provide a basis of sound empirical evidence for the requirements and the notation.

A three level notation for computational mechanisms of interaction has been developed (Simone et al., 1994a). The first version of the notation was suggested in COMIC-Risø-3-9 in February 1994 (published in Simone et al., 1994c). The development of the notation can be described as an iterative process that led by subsequent refinements from the initial formulation to the notation described in (cf. Simone et al., 1994a)**.** This process is documented by the various versions of the working paper COMIC-Mil-3-4 (Simone et al., 1994b) and by COMIC-Mil-3-5 (Simone, 1994) where the notion of active-artifact is introduced as a means to overcome the difficulties caused by the initial definition of (computational) mechanism of interaction. Each iterative step is characterized by the exploitation of previous experiences in the use of languages for the description of distributed processes. These languages have been challenged against the progressive refinement of the requirements of the notation as identified in the field studies. The need to satisfy these requirements together with the need of providing the notation with primitives suitable to support the use of a malleable and linkable notation for mechanisms of interaction contributed to the identification of properties of the notation at the level of its expressive power as well as at the level of the notation's operational semantics. Finally, these properties have informed the selection of the (formal) languages that constitute the current version of the notation for computational mechanisms of interaction. Even if the development of the notation cannot be considered as terminated, however its expressive power is enough to describe the computational mechanisms of interaction that the field studies have highlighted.

Progress and results

This research approach allowed us to exploit the various competences of the research team and to define a common research practice — one of the valuable side-product of the current research — that produced the following results:

1. Definition of mechanisms of interaction. The concept of mechanism of interaction has been put to test by being used as a framework for in-depth field studies in several field studies of the use of symbolic artifacts for articulating distributed activities in cooperative work settings. The overall result is that it is operational, leads to interesting results, and is useful as a framework for requirements analysis with a view to designing computational mechanisms of interaction.

However this experience highlighted the need of a more operational definition of the concept of mechanism of interaction. The problem arises because we want the concept to be applicable to requirements analysis, with a view to identifying likely candidates for computational mechanisms of interaction, as well as to the design of such computational mechanisms of interaction. Thus, since the allocation of functionality between human actor and artifact will change, perhaps radically, as a result of incorporating mechanisms of interaction in computer

systems, the definition should not presume a specific allocation of functionality. To the contrary, it should span the entire range of allocation of functionality and hence of local control.

In stead of the initial definition proposed in COMIC Deliverable 3.1 (Simone and Schmidt, 1993, p. 6) we have adopted the following:

> A mechanism of interaction can be defined as a protocol that, by encompassing a set of explicit conventions and prescribed procedures and supported by a symbolic artifact with a standardized format, stipulates and mediates the articulation of distributed activities so as to reduce the complexity of articulating distributed activities of large cooperative ensembles. Similarly, a computational mechanism of interaction is defined as a computer artifact that incorporates aspects of the protocol of a mechanism of interaction so that changes to the state of the mechanism induced by one actor can be automatically conveyed by the artifact to other actors in an appropriate form as stipulated by the protocol. (Schmidt and Simone, 1994)

With this definition, social and computational mechanisms of interaction are not conceived of as different kinds of mechanisms. To the contrary, all mechanisms of interaction are fundamentally and inexorably "social" mechanisms of interaction in that they are constituted by a set of procedures and conventions and supported by "a symbolic artifact with a standardized format". The adjective 'social' is redundant. Accordingly, computational mechanisms of interaction are conceived of as a special category of mechanisms of interaction that is characterized by a specific allocation of functionality between human actors and artifact.

2. Architecture of a common notation. In the course of the research under Task 3.2, it has been established that *a common computational notation for constructing different types of mechanisms of interaction is feasible*. More than that, *a three level architecture based on such a notation has been designed* (Simone et al., 1994a). It has also been demonstrated that, with the suggested three-level architecture, the notation in principle makes it possible to construct mechanisms of interaction that are malleable to any degree deemed appropriate for any particular setting.

Finally, as part of the development of the concept of mechanisms of interaction and of the definition of a notation fulfilling the requirement of malleability, a clear distinction has been achieved between the notation and the mechanism of interaction. The notation is a linguistic framework made of objects, attributes, relations, primitives etc. that is organized in three levels. Each level is characterized by a specialized set of linguistic features and provides the subsequent one with a framework for generating protocols and their instantiation, respectively. Each (instantiated) protocol is the (fully defined) specification of the mechanism of interaction under concern.

Thus, the research in Strand 3 has achieved substantial results and has progressed far beyond the scheduled targets.

3. Requirements for C-MOI. The requirements to be satisfied by computational mechanisms of interaction, as identified in COMIC Deliverable 3.1 from September 1993 (Simone and Schmidt, 1993) have been reconsidered and

revised in the light of the field study experiences, the effort to model mechanisms of interaction, and the experimental conceptual design of a computational mechanism of interaction. The following point are the most important:

(a) In order to allow for implicit understanding of certain aspects of articulation work, incomplete and not-yet complete specification, and in order not to force actors to explicitly specify a mechanism of interaction to a larger degree than deemed necessary, the notation should provide means for handling *incomplete definitions of attributes*. That is, attributes can be left un-specified and the missing specification will then be provided, perhaps at a later stage, by another mechanism or by inference from actions taken by actors (for example, if actor A starts doing task a, he is then committed to task *a* and it may also be inferred that has assumed the role *x* defined as responsible for task *a*). The notation in principle satisfies this requirement.

(b) The second and most challenging requirement for the notation is to enable two or more mechanisms of interaction to be linked, that is, building a composite mechanism of interaction from two (several, in the general case) existing ones. We can say that linking extends the modularity of computational mechanisms of interaction from their internal structure (where it is required in order to support malleability) to their environment which is expressed in terms of the other computational mechanisms of interaction available in the wider organizational field.

The notion of linking mechanisms of interaction has become crucial in the course of the past year, not only because it allows us to conceive of organizational context in terms of interactions among mechanisms of interaction in a wider organizational field but also because it allows us to make the design of individual mechanisms more manageable since its development is modular and compositional. In fact, making mechanisms of interaction linkable makes it realistically possible to design ‘mean and lean’ computational mechanisms, each of which are built to handle one specific aspect of articulation work.

In its existing form, the notation supports linking to some extent. The full development of the notation concerning linking requires a deeper understanding of its general applicability in the real cooperative ensembles.

(c) The compositional design of the mechanisms does not prevent them from still being able to be ‘aware of’ the activities of other computational mechanisms. In fact, ‘awareness’ has been introduced as a general facility enabling objects within a mechanisms to know of the state of other objects in relevant ways. This would, for instance, make it possible for an actor to be notified of the initiation of certain activities under the jurisdiction of a particular protocol.

(d) Another form of connection among mechanisms is based on the already mentioned recursiveness. In fact, recursiveness allows one to define policies according to well established protocols in order to control (constrain, redefine, complement, etc.) other protocols. For example, the control of propagation of changes to a mechanism of interaction, identified as a fundamental requirement in

Deliverable 3.1 (Schmidt et al., 1993), can be governed by another mechanism. The same holds for the policies for the definition and assignment of roles that can be established in the organizational context (by applying suitable protocols) and that are applied at the level of articulation work when roles are assigned to actors and affect the execution of tasks.

(e) Finally, the layered structure of the notation provided the initial layout of a software architecture that conceives computational mechanisms of interaction as systems embedded on the one side in their development framework (programming environment and user interface management) and on the other one in the field of work (as represented by domain specific applications and their data structures). As for the user interface, the components of the notation provide the UI designer with a modular and compositional 'language' from which different 'UI layouts can be constructed to meet the users and/or the organization requirements. First of all, the notation provides the UI designer with a set of objects at the appropriate semantic level for which it makes sense to look for an adequate graphical/multi-media representation able to support the same degree of modularity characterizing the notation. Secondly, the visibility of the notation can be tailored to different roles in different application domains. In this way each user can exploit the level of visibility her role is guaranteeing for the design of the most appropriate layout of her working space by exploiting the various primitives and the linking function.

As for the field of work, the notation specifies in a clear way where and which information could be imported from the domain specific applications. This allows for the definition of an interface that governs the implementation of the interoperability between the C-MOI supporting the articulation work and the applications supporting the work that is the object of the articulation.

Relations to other strands

The above points demonstrate the links between the results achieved in Task 3.2 with the research conducted in the other Strands. The notion of (computational) mechanisms of interaction and the possibility of linking mechanisms support an operational definition of Organizational Context, the main focus of Strand 1. The architecture underpinned by the notation can be embedded in the architecture defined in Strand 4, both in respect to the Shared Objects and the related awareness feature and in respect to the User Interface topics naturally connected with the possibility provided by virtual reality tools. Finally, malleability and linkability realized in a computational framework provide system development, the focus of Strand 2, with a framework which plays the role of a sort of 'common language' supporting and improving the interaction among the different involved actors.

The structure of the deliverable

The structure of Deliverable 3.3 reflects the research attitude mentioned above by presenting the various results grouped according to the focus on requirements, empirical studies and finally the notation.

Deliverable 3.3 is divided into three chapters and three appendices.

Chapter 1 summarizes the requirements for the notation for computational mechanisms of interaction that have derived in the course of the empirical and theoretical investigations conducted in Task 3.1.

Chapter 2 presents a requirements specification for and the conceptual design of a particular computational mechanism of interaction (‘the bug report’) based on field studies.

Chapter 3 presents the three-level architecture for a notation for constructing malleable and linkable computational mechanisms of interaction.

In addition, the appendices present three papers that supplements the development and the implementation of the notation from different approaches:

Appendix 1 proposes a formal basis for the description of dynamically modifiable objects, and explore its applicability in CSCW by exposing it to examples of different degrees of complexity.

Appendix 2 explores how the Shared Object Services and the Shared Interface Services as developed COMIC Strand 4 may be applicable to the design and incorporation of mechanisms of interaction.

Appendix 3 describes a tool which can be used to demonstrate cooperative systems. The research objective is connected with the primitive operations and their relationship with the notation. The practical purpose is to support quick and easy creation of demonstrations of cooperative interfaces in multiple workstation.

The formalism proposed in Appendix 1 is currently under investigation as one of the basic elements to be added to the notation. The framework and examples provided in this Appendix constitute a valuable starting point of a research track aimed at improving the malleability of the computational mechanisms of interaction at the level of articulation work.

Appendix 2 shows how the concepts for characterizing the notation for computational mechanisms of interaction can be incorporated as a basic constituent of a CSCW architecture. This initial structure can be further improved in order to achieve one of the aims of Task 3.2, namely the definition of the requirements of an architecture supporting the design of computational mechanisms of interaction.

Finally, Appendix 3 provides a tool that supports the interaction between users and designers during the delicate phase of requirements definition by a direct experience of a potential solution. This tool can improve the understanding of the role of mechanisms of interaction within a real cooperative setting and then stimulate the design team and the users in the identification of features and

functionalities to be added to the current version of the notation for computational mechanisms of interaction.

Future challenges

The results so far achieved within COMIC Strand 3 and their side products can be rephrased as follows:

(1) a methodological approach for the development of a design environment targeted to a specific class of systems: namely, the virtuous circle: definition of a model --> check through field studies ---> formalization into a notation, and back;

(2) a way to relate the requirements of a class of systems to the properties of the notation for their design (modularity, recursiveness, reactivity);

(3) an architecture where the target system is defined as an embedded system which shows a precise interface to (other) applications of the field of work as well as to its development environment;

(4) a framework where a relevant part of the system life cycle can be viewed as a distributed activity in which the notation plays the basic role of communication language among the various involved actors.

Therefore the objectives of a long terms research could be:

(1) the transformation of a feasibility study (the notation and implementation of the demonstrators) into a pre-engineered environment in which the various aspects of the notation are fully specified and implemented;

(2) the evaluation of to what the extent the conceptual, methodological, and notational framework described above can be applied and adapted to different classes of CSCW systems;

(3) the definition of the requirements of a CSCW platform suitable to support the (pre-)engineering of the generalized framework mentioned in the previous point.

The work of the next year will be developed at different levels.

1. At the level of requirements. While malleability is quite well-understood, linkability needs further development. Formal language theory and Object Oriented approaches provide means to represent it: what is needed is to understand which types of linkability are needed in the field in order to adapt the current tentative notation or modify it accordingly.

The requirement of supporting propagation of changes to protocols has been considered partially, mainly as triggers towards the user interface. These triggers are too simple in the sense that they could convey more information about the change in question. For example, is there a policy governing the notification of a particular change? Who is defining it? The notification could support understanding the impacts of the change on the behavior of the receiver or of its

closed environment. This is a support on how to react to the notification in an way that is aware of a common understanding of the change and of its impacts.

2. At the level of the model. The two reference dimensions of articulation work, namely time and space need to be reconsidered from scratch. In particular, we need to understand which use of the temporal dimension supports articulation work in the real cooperative ensembles. The new set of requirements coming from the field studies and the comparative analysis on how time is represented in languages to describe distributed processes will lead to the definition of a suitable metrics and associated semantics to enrich the notation with the needed features. The same approach will be taken for the dimension of space.

3. At the level of notation. The checks of the notation against the requirements progressively arising from the field studies will be prosecuted as a basic method for the development of the notation.

In any case, the current notation needs to be refined and completed in relation to different aspects:

(1) The various triggers introduced in the components of the notation to 'implement' awareness have to be more deeply considered in order to have a uniform treatment and a possible simplification of the notation.

(2) Moreover, a crucial point concerns how the various triggers combine their behavior (operationally) when awareness is combined with recursiveness and linking.

(3) Recursiveness needs a deeper understanding since it is widely used to improve the expressive power of the notation: presently, it is viewed operationally as a call for a C-MOI at the appropriate level as a co-routine. This tentative view might turn out to be insufficient when compared with the way in which recursiveness is used in the field. This aspect is fundamental as it affects the feasibility of the implementation of the notation from a complexity point of view.

(4) The capability of managing incomplete information deserves a further analysis in order to identify the inference rules that must be applied to reconstruct the missing information. Distributed AI provides techniques for implementing this feature of the notation: the crucial point is however to decide which type of 'reasoning' can be reasonably delegated to a C-MOI when the job under concern is distributed articulation work.

(5) Finally, the primitives supporting the use of the notation need to be specified in terms of the information they take as parameters so as to define which functionalities they are supporting in a realistic way: e.g., which type of local or permanent modifications, which kind of animation or simulation, and so on. Here the difficulty concerns the appropriate balance between the richness of the support the primitive is providing and the difficulty in using and interpreting the related information.

4. At the level of the demonstrator. The main goal of the demonstrator is to check the computational feasibility of the layered and modular structure of the notation. Implementing the demonstrator will concern a selection of basic

elements and of operational semantics to be put at work for the design of C-MOI according to the requirements stated in the field studies. The demonstrator will exploit existing algorithms and theories in the realization of the primitives supporting the use of the notation.

References

COMIC: *The COMIC Project (Computer-based Mechanisms of Interaction in Cooperative Work). Esprit Basic Research Project No 6225. Technical Annexe*, Lancaster University, 27 April, 1992.

Schmidt, Kjeld, and Carla Simone: *Defining Mechanisms of Interaction*, COMIC Working Paper, Risø National Laboratory, (version 1.0), 31 August, 1994. [COMIC-Risø-3-20].

Schmidt, Kjeld, Carla Simone, Peter Carstensen, Betty Hewitt, and Carsten Sørensen: "Computational Mechanisms of Interaction: Notations and Facilities," in C. Simone and K. Schmidt (eds.): *Computational Mechanisms of Interaction for CSCW*, Computing Department, Lancaster University, Lancaster, UK, 1993, pp. 109-164. - [COMIC Deliverable 3.1. Available via anonymous FTP from ftp.comp.lancs.ac.uk].

Simone, Carla: *Active Artifacts And Links To The Field Of Work In The Notation For Social And Computational MOIs*, Dipartimento di Scienze dell'Informazione, (Version 1.0), April, 1994. [COMIC-MlL-3-5].

Simone, Carla, Monica Divitini, and Alberto Pozzoli: "Towards an Architecture based on a Notation for Mechanisms of Interaction," in C. Simone and K. Schmidt (eds.): *A Notation for Computational Mechanisms of Interaction*, COMIC, Esprit Basic Research Project 6225, Computing Department, Lancaster University, Lancaster, UK, 1994a. - [This volume].

Simone, Carla, Monica Divitini, and Alberto Pozzoli: *Towards an Architecture based on a Three Level Notation for Mechanism of Interaction*, Dipartimento di Scienze dell'Informazione, (Version 2.0), April, 1994b. [COMIC-MlL-3-4].

Simone, Carla, and Kjeld Schmidt (eds.): *Computational Mechanisms of Interaction for CSCW*, COMIC, Esprit Basic Research Project 6225, Computing Department, Lancaster University, Lancaster, UK, 1993. - [COMIC Deliverable 3.1. Available via anonymous FTP from ftp.comp.lancs.ac.uk].

Simone, Carla, Kjeld Schmidt, Betty Hewitt, and Alberto Pozzoli: *An Architechture for Malleable and Linkable Mechanisms of Interaction*, Working Papers in Cognitive Science and HCI, Centre for Cognitive Science, Roskilde University, DK-4000 Roskilde, Denmark, 1994c. [WPCS-94-6].

Chapter 1

Computational mechanisms of interaction

Requirements for a general notation

Kjeld Schmidt

Risø National Laboratory

Introduction[1]

In the design of conventional computer-based systems for work settings the core issues have been to develop effective computational models of pertinent structures and processes in the field of work (data flows, conceptual schemes, knowledge representations) and adequate modes of presenting and accessing these structures and processes (user interface, functionality). While these systems, more often than not, were used in cooperative work settings and even, as in the case of systems that are part of the organizational infrastructure, were used by multiple users (e.g., database systems), the issue of *supporting the articulation of cooperative work by means of such systems* has not been addressed directly and systematically, as an issue in its own right. If the underlying model of the structures and processes in the field of work was 'valid', it was assumed that the articulation of the distributed activities was managed 'somehow'. It was certainly not a problem for the designer or the analyst.

Consider, for example, the booking system of an airline. It is a computer-based system for the cooperative task of handling reservations. The database of the booking system embodies a model of the seating arrangements of the different flights. Taken together, the seating arrangements and the database model constitutes what we can call the common field of work of the booking agents. Thus, the operators of the booking agencies cooperate by changing the state of the field of work, in casu, by reserving seats. Apart from providing a rudimentary access control facility, the booking system does not in any way support the coordination and integration of the interdependent activities of the operators. In this case, however, the field of work can be handled as a system of discrete and extremely simple (binary) state changes. Apart from the fact that a seat can only be assigned to one person at a time, there are no interactions between processes. Accordingly, even though a booking system does not support articulation work, it is seemingly

1 This paper summarizes the conceptual framework and the derived requirements for computational mechanisms of interaction as presented i COMIC Deliverable 3.2.

quite sufficient for the job. However, some if not most cooperative work arrangements in modern industrial societies are faced with a far more complex field of work in terms of number of interacting processes and states, irreversibility, concurrency, uncertainty, and so forth. Since such arrangements therefore are faced with more complex interdependencies between individual activities, they cannot rely on accomplishing their cooperative effort merely by changing the state of the field of work; such arrangements must articulate the distributed activities of their members by other means.

CSCW can be conceived of as an endeavor to understand the nature and support requirements of cooperative work arrangements with the objective of designing computer-based technologies for such arrangements (Schmidt and Bannon, 1992).

Thus, in order to be able to conceptualize and specify the support requirements of cooperative work we need to make a fundamental analytical distinction between (a) cooperative work activities in relation to the state of the field of work and mediated by changes to the state of the field of work and (b) activities that arise from the fact that the work requires and involves multiple agents whose individual activities need to be coordinated, scheduled, meshed, integrated etc. — in short: *articulated*. This distinction is fundamental to CSCW (Schmidt, 1994b).

1.1. Computational mechanisms of interaction: Definition

Cooperative work is constituted by multiple actors who are interdependent in their work and who therefore have to divide, allocate, coordinate, schedule, mesh, interrelate, integrate, etc. — in short: *articulate* their individual activities: Who is doing what, where, when, how, by means of which, under which constraints? (Strauss, 1985; Gerson and Star, 1986; Strauss, 1988).

Now, because multiple actors are involved, cooperative work is characterized by an inexorable and inescapable aspect of distributed decision making, not only in the usual sense that activities are distributed in time and space, but also — and more importantly — in the sense that the actors are semi-autonomous in terms of strategies, heuristics, conceptualizations, goals, motives, etc. (Schmidt, 1991a; Schmidt, 1991b).

The distributed character of cooperative work varies depending on a number of factors, e.g., the distribution of activities in time and space, the number of participants in the cooperative ensemble, the structural complexity posed by the field of work (interactions, heterogeneity), the degree and scope of specialization, the apperceptive complexity posed by the field of work and hence the variety of heuristics involved, and so on. The more distributed the activities of the cooperative work arrangement, the more complex the articulation of the activities of that arrangement.

With low degrees of complexity, the articulation of cooperative work can be achieved by means of the modes of interaction of everyday social life. In fact, under such conditions, the required articulation of individual activities in cooperative work is managed *so* effectively and efficiently by our repertoire of intuitive interactional modalities that the distributed nature of cooperative work is not apparent — most of the time. As demonstrated by the body of rich empirical studies of cooperative work, actors tacitly monitor each other; they make their activities sufficiently apperceptible for others; they take each others' past, present and prospective activities into account in planning and conducting their own work; they gesture, talk, write to each other, and so on (Harper et al., 1989; Heath and Luff, 1992; Harper and Hughes, 1993; Heath et al., 1993). Accordingly, much of the research in CSCW has focused on the 'shallow' support strategy of providing enhanced means of communication (desk top audio and video, shared displays, etc.), either in order to enable actors to cooperate more effectively and efficiently in spite of geographical distance, or in order to widen the repertoire of communication facilities.

However, with the complex work environments that characterize modern industrial and administrative organizations, the task of articulating the complexly interdependent and yet distributed activities is at a different order of complexity. Our everyday social and communication skills are far from sufficient in articulating the cooperative efforts of hundreds or thousands of actors engaged in myriads of complexly interdependent activities, perhaps concurrently, intermittently, or indefinitely.

In order to handle a high degree complexity of articulation work, and handle it efficiently, the articulation of the distributed activities of cooperative work requires 'deep' support by means of a category of symbolic artifacts which, in the context of a set of procedures and conventions, stipulate and mediate articulation work and thereby are instrumental in reducing the complexity of articulation work.

Such artifacts have been in use for centuries, of course — in the form of catalogues, time tables, routing schemas, kanban systems, classification systems in large repositories and so on. Now, given the infinite versatility of computer systems, it is our contention that computer-based mechanisms of interaction can provide a degree of visibility and flexibility to mechanisms of interaction that was unthinkable with previous technologies, typically based on inscriptions on paper or cardboard. This opens up new prospects of moving the boundary of allocation of functionality between human and artifact with respect to articulation work. The challenge is to change the allocation of functionality between human and artifact, not only so that much of the drudgery of articulation work (boring operations that have so far relied on human effort and vigilance) can be delegated to the artifact, but also, and more importantly, so that cooperative ensembles can articulate their distributed activities more effectively and with a higher degree of flexibility and so that they can tackle an even higher degree of complexity in the articulation of their distributed activities! (Schmidt et al., 1993)

As a generalization, we call these artifacts and the concomitant procedures and conventions ‘mechanisms of interaction’.

A mechanism of interaction can be defined as a protocol, encompassing a set of explicit conventions and prescribed procedures and supported by a symbolic artifact with a standardized format, that stipulates and mediates the articulation of distributed activities so as to reduce the complexity of articulating distributed activities of large cooperative ensembles.

In other words, and less condensed, to serve the purpose of reducing the complexity of articulation work, a mechanism of interaction must have the following characteristics:

(1) A mechanism of interaction is essentially a *protocol* in the sense that it is a set of explicit procedures and conventions that stipulate the articulation of the distributed activities.

(2) The stipulations of the protocol are, in part at least, conveyed by the *symbolic artifact* and they are thus persistent in the sense that they are, in principle, accessible independently of the particular moment or of the particular actor.

(3) At the same time, the symbolic artifact *mediates* the articulation of the distributed activities in the sense that any change to the state of execution of the protocol is conveyed to other actors in some form by means of changes to the state of the artifact.

(4) The symbolic artifact has a *standardized format* that reflects pertinent features of the protocol and thus provides affordances to and impose constraints on articulation work.

(5) The state of the artifact is *distinct* from the state of the field of work in the sense that changes to the state of the field of work are not automatically reflected in changes to the state of the execution of the protocol and, conversely, that changes to the state of the execution of the protocol are not automatically reflected in changes to the state of the field of work.

Now, in COMIC Deliverable 3.1, the concept of mechanisms of interaction was defined somewhat differently:

> “A mechanism of interaction can be defined as a device for reducing the complexity of articulating distributed activities of large cooperative ensembles by *stipulating and mediating* the articulation of the distributed activities.”(Simone and Schmidt, 1993, p. 6)

A note on the change of the definition may therefore be called for.

During the last year, the initial definition of the concept of mechanisms of interaction has been put to test by being used in several field studies of the use of symbolic artifacts for articulating distributed activities in cooperative work settings — and has turned out to create certain problems.

In fact, when applying this definition to the various artifacts used for articulation work, none of the artifacts qualified as mechanisms of interaction according to the definition, perhaps with the exception of the kanban system. The problem is that the initial definition defined a mechanism of interaction *as an artifact* with a

certain function and certain concomitant characteristics and that the artifacts that was analyzed, and probably all conventional paper-, cardboard-, and plastic-based artifacts used for these purposes, rely heavily on human actors to enact the procedures and conventions as well as to take the utterly inert artifact through all state changes. In other words, the initial definition presumed *a specific allocation of functionality* between human actor and artifact, in the form of activeness on the part of the artifact, that can only be realized by computational mechanisms of interaction. Even in the case of the kanban system, which comes very close to an allocation of functionality in which the artifact is experiences as *actively* stipulating and mediating articulation work, all state changes to the system requires human intervention for every tiny step (taking the card, reading it, interpreting it, sending it to the correct work station etc.).

As a definition of *computational* mechanisms of interaction, the initial definition has proved quite adequate, witness the comparative analysis of existing CSCW systems in Part 3 of Deliverable 3.1. (Simone and Schmidt, 1993). The problem has arisen because we also want the concept to be applicable to requirements analysis, with a view to identifying likely candidates for computational mechanisms of interaction, as well as to the design of such computational mechanisms of interaction and the underlying architecture. Thus, since the allocation of functionality between human actor and artifact will change, perhaps radically, as a result of incorporating mechanisms of interaction in computer systems, the definition should not presume a specific allocation of functionality. To the contrary, it should span the entire range of allocation of functionality and hence of local control.

The change is that the mechanism is now defined as a *protocol*, as opposed to an artifact, that is a set of procedures and conventions which are supported by an artifact with certain necessary characteristics. In so far as the term ‘mechanism’ merely denotes procedures and conventions as opposed to a tangible artifact, it is used metaphorically. This is justified, however, on the grounds that the point of the whole exercise is to design computer artifacts and for this purpose we need the term ‘mechanism’ to denote the range of different allocations of functionality between actor and artifact from almost total reliance on human intervention to almost fully automated computer artifacts.

With the new definition, social and computational mechanisms of interaction are not conceived of as different kinds of mechanisms. To the contrary, all mechanisms of interaction are fundamentally and inexorably ‘social’ mechanisms of interaction in that they are constituted by a set of procedures and conventions and supported by “a symbolic artifact with a standardized format”. The adjective ‘social’ is redundant.

Accordingly, computational mechanisms of interaction are conceived of as a special category of mechanisms of interaction that is characterized by a specific allocation of functionality between human actors and artifact:

A computational mechanism of interaction is a computer artifact that incorporates aspects of the protocol of a mechanism of interaction so that changes to the

state of the mechanism induced by one actor can be automatically conveyed by the artifact to other actors in an appropriate form as stipulated by the protocol.

1.2. Mechanisms of interaction: The experience

The aim of providing structured support for the articulation of distributed activities is shared by many researchers in the CSCW community. However, most of the mechanisms of interaction in existing CSCW systems are experienced as excessively rigid, either because the embedded mechanism of interaction is not accessible to the actors and cannot be changed, e.g., THE COORDINATOR (Winograd and Flores, 1986; Flores et al., 1988), or because the facilities for changing the mechanism do not support respecification of the mechanism by the actors themselves, at the semantic level of articulation work, e.g., DOMINO (Kreifelts et al., 1991a; Kreifelts et al., 1991b). By contrast, in view of the situated character of cooperative work, mechanisms of interaction should be conceived of as local and temporary closures (Gerson and Star, 1986; Schmidt, 1991b; Schmidt, 1994a). Our primary objective is thus to support actors in accessing and manipulating the mechanism of interaction in such a way that they are able to handle contingencies and adapt the stipulations of the mechanisms to changing requirements in their environment.

Now, a number of recent research projects attempt to make CSCW applications flexible at the level of articulation work, e.g., EGRET (Johnson, 1992) and CONVERSATIONBUILDER (Kaplan et al., 1992).

The CONVERSATIONBUILDER was explicitly developed as a support tool for providing flexible active support for cooperative work activities. This flexibility is achieved by providing "appropriate mechanisms for the support of collaboration rather than specific policies. Policies can be built out of mechanisms, if the right mechanisms are provided" (Bogia et al., 1994).

On the other hand, EGRET supports cooperative work a dynamically changing environment by giving users the ability to develop new 'schemas' which can include tasks, descriptions, names, etc. EGRET keeps track of the variation in the schemas by providing facilities for listing deviations from each schema. This enables all group members to see, for example, the number and types of solutions to a particular problem. Once consensus has been achieved, the database can once again be made consistent and all members then have new consistent schemas. Some degree of local control of execution and visibility of changes are supported. All changes are propagated throughout the database.

While closely related, our approach differs in one important respect, namely in the attempt to develop a general notation that is comprehensive enough to specify any mechanism of interaction and which at the same time supports the specification of mechanisms of interaction in terms of articulation work, by the actors themselves in a cooperative manner.

Since mechanisms of interaction are local and temporary closures, no mechanism will be able to handle all aspects of articulation work in all work domains. Particular mechanisms of interaction will invariably be designed to support cooperating actors in particular aspects of articulation work that are particularly complex and possibly even domain-specific. Thus, in order not to create artificial barriers between different aspects of articulation work, computational mechanisms of interaction must be designed in such a way that they can be linked to other mechanisms, locally and temporarily.

A computational mechanism of interaction should thus be conceived of as an abstract device incorporated in a particular software application (e.g., a CASE tool, an office information system, a CAD system, a production control system, etc.) so as to support the articulation of the distributed activities of multiple actors with respect to that application — without imposing on actors an undue impedance between articulation work and work. As abstract devices, mechanisms of interaction are intended to facilitate the design of domain-specific applications *in such a way* that they support the fluid interrelation of articulation work with respect to the *multiple applications* required to do the work in a particular setting, without imposing an impedance on the articulation of cooperative work with respect to different applications — for instance, in the case of mechanical design: with respect to project management tools, fault management facilities, CAD tools, process planning systems, classification schemes for common repositories (of previous designs, components, work in progress, drawings, patents), and so on. To avoid creating such an impedance between the articulation of cooperative work with respect to different applications, a general notation for specifying linkable mechanisms of interaction is required.

A different and more general approach was taken by Malone et al. with OVAL (Malone et al., 1992). The main purpose of OVAL is to provide basic building blocks for integration of many types of information and applications, and at the same time provide users with the ability to tailor this integration. OVAL is thus an important contribution regarding notations for constructing mechanisms of interaction. Contrary to the notations in the CONVERSATIONBUILDER and the EGRET Framework, by providing a set of basic primitives such as objects, views, agents, and links, the notation in OVAL (the specification language) is intended to be a general notation for designing CSCW systems. The notation provided by OVAL is thus very abstract and flexible. The notation is problematic, however, in the sense that basic primitives provided by the notation are not at the semantic level of articulation work. By contrast, the aim of the present approach is to support actors in specifying and respecifying the mechanism of interaction in terms of their everyday cooperative activities.

1.3. Requirements for computational mechanisms of interaction

The concept of mechanisms of interaction has been developed as a generalization of phenomena described in different ways in different empirical investigations of the use of artifactually embodied protocols for the articulation of cooperative activities in different work domains:

- standard operating procedures in administrative work (Zimmerman, 1966; Zimmerman, 1969b; Zimmerman, 1969a; Wynn, 1979; Suchman, 1983; Suchman and Wynn, 1984; Wynn, 1991);
- classification schemes for large repositories (Bowker and Star, 1991; Andersen, 1994; Sørensen, 1994c);
- checklists (Degani and Wiener, 1990);
- time tables in urban transport (Heath and Luff, 1992);
- flight progress strips in air traffic control (Harper et al., 1989; Harper and Hughes, 1993);
- production control systems in manufacturing (Schmidt, 1994b);
- planning tools for manufacturing design (Bucciarelli, 1988; Sørensen, 1994a; Carstensen et al., 1995);
- fault correction procedures in manufacturing and software design (Carstensen, 1994; Carstensen et al., 1994; Pycock, 1994; Pycock and Sharrock, 1994; Sørensen, 1994b).

In addition, empirical studies of CSCW systems in use have of course also contributed to the development of the concept of mechanisms of interaction (e.g., Schäl, 1994).

From the evidence of these and other studies, we can stipulate (Schmidt et al., 1993) a set of general requirements for computational mechanisms of interaction and, by implication, for a general notation for specifying mechanisms of interaction.

3.1. Malleability and linkability

From the evidence of the corpus of empirical studies of uses of artifactually embodied protocols for articulating cooperative activities, we have derived a set of general requirements for computational mechanisms of interaction and, by implication, for a general notation for constructing computational mechanisms of interaction (cf. Schmidt et al., 1993; Simone et al., 1994).

First and foremost, since mechanisms of interaction, as plans in general, are “resources for situated action” (Suchman, 1987), a mechanism of interaction must be *malleable* in the sense that it supports users in specifying its behavior.

(1) Global and lasting changes. Since organizational demands and constraints change, it should be possible for actor to design and develop new mechanisms of interaction and to make lasting modification to existing ones. In the case of the

bug form mechanism, for example, the entire mechanism — the artifact as well as the procedures and conventions — was designed from scratch by the actors themselves. Accordingly:

> A computational mechanism of interaction must provide facilities for actors to specify and respecify the behavior of the mechanism so as to enable actors to meet changing organizational requirements.

(2) Local and temporary changes. Actors must be able to make local and temporary changes to the behavior of the mechanism, for instance by suspending or overruling a step, by 'rewinding' a procedure, escaping from a situation, or even restarting the mechanism from another point. That is:

> A computational mechanism of interaction must give actors control of the execution of the mechanism so as to cope with unforeseen contingencies.

In making such specifications, actors may not be able to completely specify the behavior of the mechanism or they may prefer not to specify certain attributes explicitly or to defer their specification:

(3) Partial definitions. While a mechanism of interaction cannot be constructed without explicit conventions and prescribed procedures, mechanisms of interaction are, in principle, under-specified (Suchman, 1983; Suchman and Wynn, 1984; Suchman, 1987). Protocols are, to some extent, only specified in the course of the work. Furthermore, a protocol and the whole equipage of ensuing stipulations can be invoked implicitly, without any explicit announcements, for instance by certain actors taking certain actions (Strauss, 1985; Schäl, 1994).

Thus, in order to allow for implicit understanding of certain aspects of articulation work as well as incomplete and not-yet complete specification, and also in order not to force actors to explicitly specify a mechanism of interaction to a larger degree than deemed necessary, the mechanism should provide means for handling *partial specifications of attributes*. That is, it should be possible for attributes to be left un-specified and for the missing specification to be provided, perhaps at a later stage, by another mechanism or by inference from actions taken by actors. For example, if actor A starts performing task *a*, he or she may then be taken to be committed to accomplish task *a* and it may also be inferred that he or she has assumed the role *x* defined as responsible for task *a*. Hence,

> A computational mechanism of interaction must provide avenues for attributes to be left un-specified and for the missing specification to be provided, at some point, by another mechanism or by inference from actions taken by actors.

(4) Visibility. From these requirements (1-3) follows that the specification of the behavior of the mechanism must be 'visible' to actors, not only in the sense

that it is accessible but also, and especially, that it *makes sense* to actors as specifications in terms of articulation work:

> In order for actors to be able to exercise their control of the execution of the mechanism and respecify the behavior of the mechanism, the specification of the behavior of the mechanism must be accessible and manipulable to actors and, more specifically, accessible and manipulable *at the semantic level of articulation work.*

Since the specification of the behavior of the mechanism must be 'visible' to actors *at the semantic level of articulation work*, the objects and functional primitives offered by the mechanism must be expressed in terms of operations of articulation work with respect to roles, actors, tasks, activities, conceptual structures, resources, and so on (cf. the model of articulation work in Figure 15).

(5) Control of propagation of changes. Since articulation work is a recursive function (Gerson and Star, 1986), changing a mechanism of interaction may be done cooperatively, as part and parcel of the cooperative effort. That is, it should be possible to change the mechanism of interaction while it is running, without having to suspend all activities within the cooperative ensemble for some time:

> A computational mechanism of interaction must provide means for dynamic reconfiguration of the protocol and, accordingly, give actors means of controlling the propagation of changes to the specification of the behavior of the mechanism.

(6) Relating to the field of work. Since mechanisms of interaction are 'local and temporary closures' (Gerson and Star, 1986), no single mechanism will apply to all aspects of articulation work in all domains of work. Accordingly, a computational mechanism of interaction is to be conceived of as a specialized software device that is distinct from the state of the field of work and yet embedded in an application so as to support the articulation of the distributed activities of multiple actors with respect to the field of work as represented by that application.

> As an embedded system, a computational mechanism of interaction must provide means of identifying pertinent features of the field of work as represented by the data structures and the functionality of the application in which it is embedded.

(7) Linkability. Since no single mechanism will apply to all aspects of articulation work in all domains of work, the computational mechanism of interaction must provide means for establishing links with other computational mechanisms of interaction embedded in other applications:

A computational mechanism of interaction must provide facilities for establishing links to other mechanisms of interaction in the wider organizational field.

3.2. Objects and operations of articulation work

A general notation for mechanisms of interaction should provide potential support for articulation work with respect to the different "salient dimensions" of articulation work (Strauss, 1985).

A more elaborate 'inventory' of the objects and operations of articulation work could look as follows:[1]

On one hand, the distributed activities of a cooperative work arrangement are articulated with respect to objects pertaining to *the cooperative work arrangement* itself, that is:

- Articulation in terms of **roles**, that is, in terms of general responsibilities for classes of tasks and resources.
- Articulation in terms of **actors**, that is, the committed or actual participants in the cooperative effort in question (in different capacities such as roles, jobs, individuals, collectives): Who is committed to do what when? Who is doing this?
- Articulation in terms of **human resources**, that is the potential participants in the cooperative effort in question: Which partners are potentially relevant for a particular project in terms of skills, competing commitments etc.? Who are available when?
- Articulation in terms of **tasks**, that is, in terms of an operational intention (goals to attain, obligations and commitments to meet): What is the problem? What is to be done? Who should do it? Should I do it? Which task is (normally, advisably, or according to statute) to be undertaken in which circumstances, by which actor, based on what information and which criteria, creating what information? What is the (normal, advisable, or statutory) relation between different tasks (procedure, workflow)?
- Articulation in terms of **activities**, that is, in terms of an unfolding course of purposive action. What are the others doing, and why? What have they done, what will they be doing, etc.? Do they cope?[2]

---

1 A similar attempt to identify "basic coordination processes" is made by Malone and Crowston (1992).

2 The terminology used here comes from the Scandinavian tradition within software development: An *activity* is used to denote a work process as an unfolding course of action, but only those aspects of a work process that are relevant to doing the work with the currently available resources, not all other incidents that may occur in the same course of action but which are of no consequence for getting the work done (like spilling coffee). — The concept of a task, on the other hand, is used to denote an operational intent, irrespective of how it is implemented. A task is expressed in terms of *what*, an activity in terms of *how*. A task can be *accomplished*, an activity can cease. (Andersen et al., 1990)

On the other hand, the distributed activities of a cooperative work arrangement are articulated with respect to objects pertaining to *the field of work* of that cooperative work arrangement:

- Articulation in terms of the common **resources** which constitute the field of work, potentially or actually:
  - ◊ **information** resources (documents, letters, applications, notes, files, memos, reports, drawings): Which actor can access, change, delete, copy which information resources? To which actor is the object to be displayed, in which format? Which actor can see who doing what to which objects?
  - ◊ **material** resources (materials, components, assemblies). Which materials, components, assemblies are available where, when, how, in which quantity? What are their characteristics?
  - ◊ **technical** resources (tools, fixtures, machinery, software applications). What are their operational characteristics (machining tolerance, suitability for different kinds of materials and material dimensions, processing time and cost)?
  - ◊ **infrastructural** resources (rooms, buildings, communication facilities, transportation facilities). What are their operational characteristics (capacity, location, compatibility, turnaround time, bandwith)?
- Articulation in terms of **conceptual structures**, that is, in terms of the relationship between categories used within a specific community as ordering devices with respect to the field of work, either by *adopting* conceptual structures (by defining categories, i.e. by establishing prototypical, causal, genetic, historical, means/end relationships between categories), or by *applying* such conceptual structures (by classifying events, objects, etc.) so as to monitor activities with respect to, direct attention to, make sense of, act on etc. certain aspects of the state of the field of work.

As indicated, these dimensions of articulation work are interdependent. For example, articulation with respect to *tasks* may refer to a set of activities realizing a particular task, the actual actor doing the activity, various resources deployed to the activity, the categorization of the resources pertaining to the field of work.

However, in order to be able to grasp the dynamics of articulation work, we need to make another distinction, namely between *nominal* and *actual* articulation work, that is, articulation work in term of *nominal* (ideational, potential, not yet realized) or *actual* (existent, definite, realized) objects and operations.

That is, the model of ‘objects’ of articulation work and the concomitant ‘operations’ can be ordered along two axes, as in Figure 15:

(1) On the one hand we distinguish articulation work according to its status, that is, *nominal* and *actual*.

(2) On the other hand we distinguish articulation work with respect to the component parts of the *cooperative work arrangement* and the objects and processes of *the field of work*.

Nominal		**Actual**	
Objects of articulation work	**Operations with respect to objects of articulation work**	**Objects of articulation work**	**Operations with respect to objects of articulation work**
Articulation work with respect to the cooperative work arrangement			
Role	assign to [Committed actor]; responsible for [Task, Resource]	Committed-actor	assume , accept, reject [Role]; initiate [Activity];
Task	point out, express; divide, relate; allocate, volunteer; accept, reject; order, countermand; accomplish, assess; approve, disapprove; realized by [Activity]	Activity	[Committed actor] initiate; [Actor-in-action] undertake, do, accomplish; realize [Task]; [Actor-in-action] makes publicly perceptible, monitors, is aware of, explains, questions;
Human resource	locate, allocate, reserve;	Actor-in-action	initiates [Activity]; does [Activity];
Articulation work with respect to the field of work			
Conceptual structures	categorize: define, relate, exemplify relations between categories pertaining to [Field of Work];	State of field of work	classify aspect of [State of field of work]; monitor, direct attention to, make sense of, act on aspect of [State of field of work];
Informational resource	locate, obtain access to, block access to;	Informational resources-in-use	show, hide content of; publicize, conceal existence of;
Material resource	locate, procure; allocate, reserve to [Task];	Material resources-in-use	deploy, consume; transform;
Technical resource	locate, procure; allocate, reserve to [Task];	Technical resources-in-use	deploy; use;
Infrastructural resource	reserve;	Infrastructural resources-in-use	use;

Figure 1-1. The ‘articulation work model’. The table identifies the elemental objects of articulation work and gives examples typical elemental operations on these objects.

The notation for constructing mechanisms of interaction should provide a set of *elemental operations* that can be invoked with respect to the objects of articula-

tion work. The diagram of figure 1-2 depicts the objects of articulation work and the elemental operations on them.

It is pressumed that the set of elemental operations identified here are sufficient for our purpose. Whether that is in fact the case, however, is an empirical question (as everything else in this line of research) and is currently being assessed.

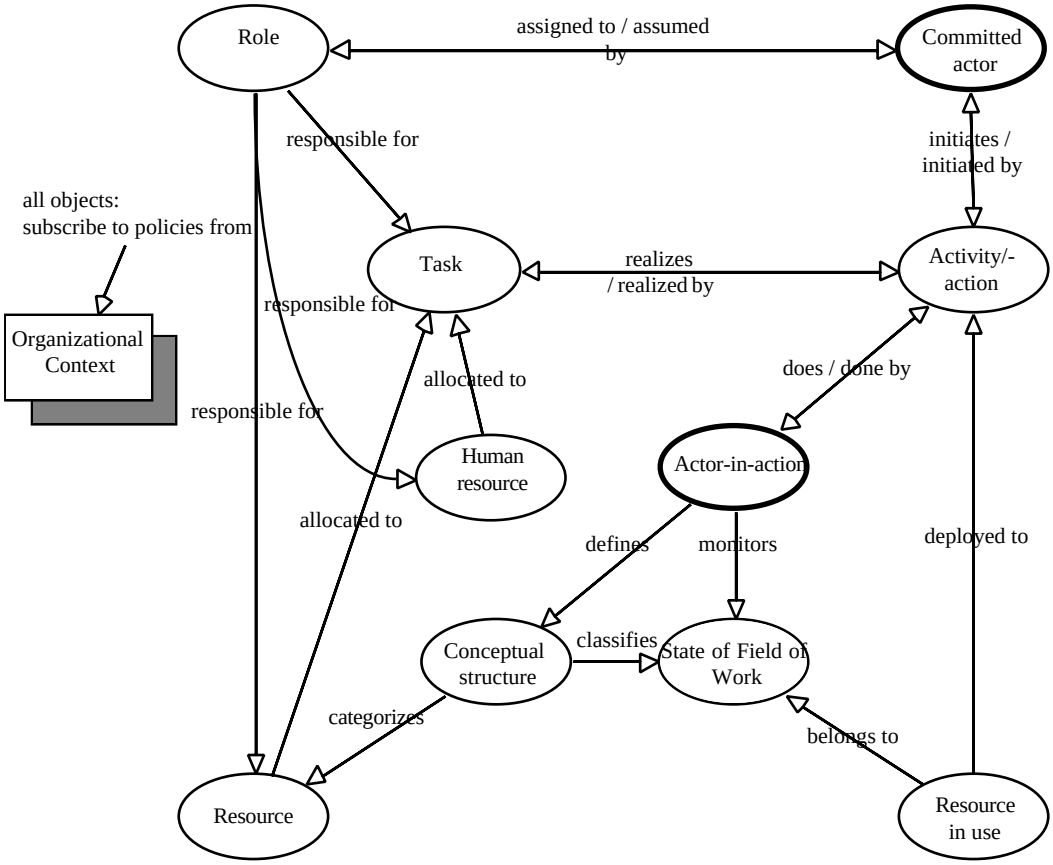


Figure 1-2. Objects and elemental operations of articulation work. The objects on the left hand side of the diagram are of *nominal* status, whereas the objects on the right hand side are of *actual* status. The 'missing links' between objects can be constructed indirectly, by creating composite operations.

References

Andersen, Hans H. K.: "Classification schemes: Supporting articulation work in technical documentation," in H. Albrechtsen (ed.): *ISKO '94. Knowledge Organisation and Quality Management, Copenhagen, Denmark, June 21-24, 1994*, 1994.

Andersen, N. E., F. Kensing, J. Lundin, L. Mathiassen, A. Munk-Madsen, M. Rasbech, and P. Sørgaard: *Professional Systems Development — Experience, Ideas, and Action*, Prentice-Hall, Englewood-Cliffs, New Jersey, 1990.

Bogia, Douglas P., William J. Tolone, Celsina Bignoli, and Simon M. Kaplan: “Issues in the Design of Collaborative Systems: Lessons from ConversationBuilder,” in D. Shapiro, M. Tauber, and R. Traünmuller (eds.): *The Design of Computer-Supported Cooperative Work and Groupware Systems*, Elsevier Science, Amsterdam, 1994. - Forthcoming.

Bowker, Geoffrey, and Susan Leigh Star: “Situations vs. Standards in Long-Term, Wide-Scale Decision-Making: The Case of the International Classification of Diseases,” in J. F. Nunamaker, Jr. and R. H. Sprague, Jr. (eds.): *Proceedings of the Twenty-Fourth Annual Hawaii International Conference on System Sciences, Kauai, Hawaii, January 7-11, 1991*, IEEE Computer Society Press, 1991, vol. IV.

Bucciarelli, Louis L.: “Engineering Design Process,” in F. A. Dubinskas (ed.): *Making Time. Ethnographies of High-Technology Organizations*, Temple University Press, Philadelphia, 1988, pp. 92-122.

Carstensen, Peter: “The bug report form,” in K. Schmidt (ed.): *Social Mechanisms of Interaction*, COMIC, Esprit Basic Research Project 6225, Computing Department, Lancaster University, Lancaster, UK, 1994. - [COMIC Deliverable 3.2. Available via anonymous FTP from ftp.comp.lancs.ac.uk].

Carstensen, Peter, Tuomo Tuikka, and Carsten Sørensen: “Are We Done Now? Towards Requirements for Computer Supported Cooperative Software Testing,” in P. Kerola, A. Juustila, and J. Järvinen (eds.): *17th IRIS Seminar, Syöte, Finland, 6-9 August, 1994*, Oulu University, 1994, vol. 1, pp. 424-451.

Carstensen, Peter H., Carsten Sørensen, and Henrik Borstrøm: “Two is Fine, Four is a Mess: Reducing Complexity of Articulation Work in Manufacturing,” *COOP ’95. International Workshop on the Design of Cooperative Systems, Antibes-Juan-les-Pins, France, 25-27 January 1995*, INRIA Sophia Antipolis, France, 1995, pp. 314-333.

Degani, Asaf, and Earl L. Wiener: *Human Factors of Flight-Deck Checklists: The Normal Checklist*, National Aeronautics and Space Administration, Ames Research Center, Moffett Field, California, May, 1990. [NASA Contractor Report 177549; Contract NCC2-377].

Flores, Fernando, Michael Graves, Brad Hartfield, and Terry Winograd: “Computer Systems and the Design of Organizational Interaction,” *ACM Transactions on Office Information Systems*, vol. 6, no. 2, April 1988, pp. 153-172.

Gerson, Elihu M., and Susan Leigh Star: “Analyzing Due Process in the Workplace,” *ACM Transactions on Office Information Systems*, vol. 4, no. 3, July 1986, pp. 257-270.

Harper, Richard H. R., and John A. Hughes: “What a f-ing system! Send ’em all to the same place and then expect us to stop ’em hitting. Managing technology work in air traffic control,” in G. Button (ed.): *Technology in Working Order. Studies of work, interaction, and technology*, Routledge, London and New York, 1993, pp. 127-144.

Harper, Richard R., John A. Hughes, and Dan Z. Shapiro: *The Functionality of Flight Strips in ATC Work. The report for the Civil Aviation Authority*, Lancaster Sociotechnics Group, Department of Sociology, Lancaster University, January, 1989.

Heath, Christian, Marina Jirotka, Paul Luff, and Jon Hindmarsh: “Unpacking Collaboration: the Interactional Organisation of Trading in a City Dealing Room,” in G. De Michelis, C. Simone, and K. Schmidt (eds.): *ECSCW ’93. Proceedings of the Third European Conference on Computer-Supported Cooperative Work, 13-17 September 1993, Milan, Italy*, Kluwer Academic Publishers, Dordrecht, 1993, pp. 155-170.

Heath, Christian, and Paul Luff: “Collaboration and Control. Crisis Management and Multimedia Technology in London Underground Control Rooms,” *Computer Supported Cooperative Work (CSCW). An International Journal*, vol. 1, no. 1-2, 1992, pp. 69-94.

Johnson, Philip: “Supporting Exploratory CSCW with the EGRET Framework,” in J. Turner and R. Kraut (eds.): *CSCW ’92. Proceedings of the Conference on Computer-Supported Cooperative Work, Toronto, Canada, October 31 to November 4, 1992*, ACM Press, New York, 1992, pp. 298-305.

Kaplan, Simon M., William J. Tolone, Douglas P. Bogia, and Celsina Bignoli: “Flexible, Active Support for Collaborative Work with Conversation Builder,” in J. Turner and R. Kraut (eds.): *CSCW ’92. Proceedings of the Conference on Computer-Supported Cooperative Work, Toronto, Canada, October 31 to November 4, 1992*, ACM Press, New York, 1992, pp. 378-385.

Kreifelts, Thomas, Elke Hinrichs, Karl-Heinz Klein, Peter Seuffert, and Gerd Woetzel: “Experiences with the DOMINO Office Procedure System,” in L. Bannon, M. Robinson, and K. Schmidt (eds.): *ECSCW ’91. Proceedings of the Second European Conference on Computer-Supported Cooperative Work*, Kluwer Academic Publishers, Amsterdam, 1991a, pp. 117-130.

Kreifelts, Thomas, Frank Victor, Gerd Woetzel, and Michael Woitass: “A Design Tools for Autonomous Agents,” in J. M. Bowers and S. D. Benford (eds.): *Studies in Computer Supported Cooperative Work. Theory, Practice and Design*, North-Holland, Amsterdam, 1991b, pp. 131-144.

Malone, Thomas W., and Kevin Crowston: *The Interdisciplinary Study of Coordination*, Center of Coordination Science, MIT, 1992.

Malone, Thomas W., Kum-Yew Lai, and Christopher Fry: “Experiments with Oval: A Radically Tailorable Tool for Cooperative Work,” in J. Turner and R. Kraut (eds.): *CSCW ’92. Proceedings of the Conference on Computer-Supported Cooperative Work, Toronto, Canada, October 31 to November 4, 1992*, ACM Press, New York, 1992, pp. 289-297.

Pycock, James: “Mechanisms of interaction and technologies of representation: Examining a case study,” in K. Schmidt (ed.): *Social Mechanisms of Interaction*, COMIC, Esprit Basic Research Project 6225, Computing Department, Lancaster University, Lancaster, UK, 1994. - [COMIC Deliverable 3.2. Available via anonymous FTP from ftp.comp.lancs.ac.uk].

Pycock, James, and Wes Sharrock: “The fault report form: Mechanisms of interaction in design and development project work,” in K. Schmidt (ed.): *Social Mechanisms of Interaction*, COMIC, Esprit Basic Research Project 6225, Computing Department, Lancaster University, Lancaster, UK, 1994. - [COMIC Deliverable 3.2. Available via anonymous FTP from ftp.comp.lancs.ac.uk].

Schäl, Thomal: “System design for cooperative work in the language-action perspective,” in D. Shapiro, M. Tauber, and R. Traünmuller (eds.): *The Design of Computer-Supported Cooperative Work and Groupware Systems*, Elsevier Science, Amsterdam, 1994. - Forthcoming.

Schmidt, Kjeld: “Cooperative Work. A Conceptual Framework,” in J. Rasmussen, B. Brehmer, and J. Leplat (eds.): *Distributed Decision Making. Cognitive Models for Cooperative Work*, John Wiley & Sons, Chichester, 1991a, pp. 75-109.

Schmidt, Kjeld: “Riding a Tiger, or Computer Supported Cooperative Work,” in L. Bannon, M. Robinson, and K. Schmidt (eds.): *ECSCW ’91. Proceedings of the Second European Conference on Computer-Supported Cooperative Work*, Kluwer Academic Publishers, Amsterdam, 1991b, pp. 1-16.

Schmidt, Kjeld: “Cooperative work and its articulation: Requirements for computer support,” *Travail Humain*, vol. 57, no. 4, 1994a, pp. 403-424. - [In press].

Schmidt, Kjeld: *Modes and Mechanisms of Interaction in Cooperative Work*, Risø National Laboratory, P.O. Box 49, DK-4000 Roskilde, Denmark, 1994b. [Risø-R-666(EN)].

Schmidt, Kjeld, and Liam Bannon: “Taking CSCW Seriously: Supporting Articulation Work,” *Computer Supported Cooperative Work (CSCW). An International Journal*, vol. 1, no. 1-2, 1992, pp. 7-40.

Schmidt, Kjeld, Carla Simone, Peter Carstensen, Betty Hewitt, and Carsten Sørensen: “Computational Mechanisms of Interaction: Notations and Facilities,” in C. Simone and K. Schmidt (eds.): *Computational Mechanisms of Interaction for CSCW*, Computing Department, Lancaster University, Lancaster, UK, 1993, pp. 109-164. - [COMIC Deliverable 3.1. Available via anonymous FTP from ftp.comp.lancs.ac.uk].

Simone, Carla, and Kjeld Schmidt (eds.): *Computational Mechanisms of Interaction for CSCW*, COMIC, Esprit Basic Research Project 6225, Computing Department, Lancaster University, Lancaster, UK, 1993. - [COMIC Deliverable 3.1. Available via anonymous FTP from ftp.comp.lancs.ac.uk].

Simone, Carla, Kjeld Schmidt, Betty Hewitt, and Alberto Pozzoli: *An Architechture for Malleable and Linkable Mechanisms of Interaction*, Working Papers in Cognitive Science and HCI, Centre for Cognitive Science, Roskilde University, DK-4000 Roskilde, Denmark, 1994. [WPCS-94-6].

Strauss, Anselm: “Work and the Division of Labor,” *The Sociological Quarterly*, vol. 26, no. 1, 1985, pp. 1-19.

Strauss, Anselm: “The Articulation of Project Work: An Organizational Process,” *The Sociological Quarterly*, vol. 29, no. 2, 1988, pp. 163-178.

Suchman, Lucy A.: “Office Procedures as Practical Action: Models of Work and System Design,” *ACM Transactions on Office Information Systems*, vol. 1, no. 4, October 1983, pp. 320-328.

Suchman, Lucy A.: *Plans and situated actions. The problem of human-machine communication*, Cambridge University Press, Cambridge, 1987.

Suchman, Lucy A., and Eleanor Wynn: “Procedures and Problems in the Office,” *Office: Technology and People*, vol. 2, 1984, pp. 133-154.

Sørensen, Carsten: “The augmented bill of materials,” in K. Schmidt (ed.): *Social Mechanisms of Interaction*, COMIC, Esprit Basic Research Project 6225, Computing Department, Lancaster University, Lancaster, UK, 1994a. - [COMIC Deliverable 3.2. Available via anonymous FTP from ftp.comp.lancs.ac.uk].

Sørensen, Carsten: “The CEDAC board,” in K. Schmidt (ed.): *Social Mechanisms of Interaction*, COMIC, Esprit Basic Research Project 6225, Computing Department, Lancaster University, Lancaster, UK, 1994b. - [COMIC Deliverable 3.2. Available via anonymous FTP from ftp.comp.lancs.ac.uk].

Sørensen, Carsten: “The product classification scheme,” in K. Schmidt (ed.): *Social Mechanisms of Interaction*, COMIC, Esprit Basic Research Project 6225, Computing Department, Lancaster University, Lancaster, UK, 1994c. - [COMIC Deliverable 3.2. Available via anonymous FTP from ftp.comp.lancs.ac.uk].

Winograd, Terry, and Fernando Flores: *Understanding Computers and Cognition: A New Foundation for Design*, Ablex Publishing Corp., Norwood, New Jersey, 1986.

Wynn, Eleanor: “Taking Practice Seriously,” in J. Greenbaum and M. Kyng (eds.): *Design at Work: Cooperative Design of Computer Systems*, Lawrence Erlbaum, Hillsdale, New Jersey, 1991, pp. 45-64.

Wynn, Eleanor H.: “Office Conversation as an Information Medium,” Ph. D. dissertation, University of California, Berkeley, 1979.

Zimmerman, Don H.: “Paper work and people work: A study of a public assistance agency,” Ph.D. Dissertation, University of California, Los Angeles, Los Angeles, 1966.

Zimmerman, Don H.: “Record-Keeping and the Intake Process in a Public Welfare Agency,” in S. Wheeler (ed.): *On Record: Files and Dossiers in American Life*, Russell Sage Foundation, New York, 1969a, pp. 319-354.

Zimmerman, Don H.: “Tasks and Troubles: The Practical Bases of Work Activities in a Public Assistance Agency,” in D. A. Hansen (ed.): *Explorations in Sociology and Counseling*, Houghton-Mifflin, New York, 1969b, pp. 237-266.

Chapter 2

Requirements for a computational mechanism of interaction: An example

Peter Carstensen and Carsten Sørensen

Risø National Laboratory

2.1. Introduction

Designing computer systems supporting articulation activities requires a deep and coherent understanding of the work domain to be supported. The aim of this chapter is to illustrate how requirements for a specific computer based mechanism of interaction (Schmidt et al., 1993) supporting a specific work setting could be established and expressed concisely, using empirical data from a field study as basis. For this purpose, we have chosen to discuss requirements for support of certain aspects of the articulation of software testing. Software testing is one of the areas in which we have conducted field studies (see Carstensen, 1994; Carstensen and Sørensen, 1994).

The aim of the requirements is to provide input for refining the notation for, and architecture of, the concept of computational mechanisms of interaction. The results of these refinements can be seen both in the previous and the following chapters of this book. This chapter should, therefore, not be read as a documentation of a software-development process. The central approach have not been to come up with new innovations, but rather to provide an exemplification of requirements for a computational mechanism of interaction. In real life settings, it would, in addition, be essential to address problems such as: How to identify candidates for computational mechanisms of interaction? How to draw the boundary of the computer systems? How to avoid simple automation of the existing mechanism? etc.

Software testing is a very complicated task, which, in practice, is impossible to accomplished exhaustively (Myers, 1979; Parnas, 1985). Hence, much effort is required to articulate the activities, negotiate software acceptance criteria for usability, reliability, capability, etc., and to establish consensus of when the software is acceptable. In large scale software projects, like the one reported from here, many actors with different perspectives and different areas of competence, are involved in the testing process. Plans and procedures for who is testing what, classifying errors, reporting back, handling the corrections, re-testing, etc. are needed. The involved testers and designers have to cooperate and they become mutually

interdependent in their work. This affects the complexity of articulation the testing process. Much effort is needed from the actors in order to articulate the activities, i.e., mesh, allocate, coordinate, negotiate, etc. the required tasks and activities, and relate these to actors, material resources, etc. (Schmidt, 1993; Strauss, 1985).

This chapter formulate a set of requirements for computer based mechanisms supporting the articulation work involved in registering, diagnosing, and correcting software errors. The requirements are mainly derived from the findings of a field study of a large software development project at Foss Electric.

Much research have been conducted in the software testing field (Gelperin and Hetzel, 1988), but very little of this research has addressed the organizational process and cooperative aspects of software testing. In the field of Computer Supported Cooperative Work (CSCW) much research have addressed aspects of communication among relatively few actors constantly monitoring each others activities (e.g., Heath et al., 1993; Ishii et al., 1993) or modeling the structure of the ongoing communication (e.g., Flores et al., 1988). An attempt to address the cooperative aspects of software testing is the CSRS system supporting collaborative software review (Johnson and Tjahjono, 1993). The idea was to apply an existing hypertext-based environment for building systems to keep track of the decisions taken in a group, on the field of software inspection.

Our focus has been an attempt to set up a set of requirements for a mechanism that, in a more active manner, stipulates the required work flow, and mediates the needed information among the involved actors. To do this, the requirements address two central aspects: First, which conceptualizations of the field of work and the work arrangements must be provided. And second, which facilities are needed for stipulating the central work flow, in the aspects of articulating a software testing process we address, i.e., facilities required for the articulation of registering, diagnosing, and correcting software bugs. We do not claim that tools supporting the articulation of software testing should be seen as isolated tools. They should be integrated with other tools for software testing. For the purpose of illustration, this integration has, however, been considered outside the scope of this chapter.

Requirements for how the interaction with the actors (users) can run, what the data structures embedded in the mechanism must contain, what browsing facilities must be provided, and which other computer based mechanisms the mechanism must link to are discussed, both in terms of overall requirements and in terms of more design oriented aspects.

Our approach to collecting data in order to understand the articulation aspects of the software testing process at Foss Electric has mainly been based on qualitative empirical studies. We conducted a series of approximately 10 open-ended qualitative interviews (Patton, 1980), participated in 5 integration and planning meetings, inspected a number of relevant documents and spent about 35 hours observing the software integration and testing process over a two month period towards the end of the project. The approach in the field study was inspired by Work Analysis (Schmidt and Carstensen, 1990).

The following section gives an overview of the Foss Electric company and the concrete project studied. Based on this, a set of overall requirements for computer support of the articulation of software testing is established in section 3. Section 4 describes in further details how the existing paper based mechanism for handling the bug registration and correction is functioning. The central section in this chapter is section 5 which establishes a set of detailed requirements for how to computer support the mechanism described in section 4. Section 6 concludes the chapter and discuss the generality of the requirements.

2.2. The Case study

This section attempts to establish an understanding of the cooperative work setting we have had in mind when defining the requirements discussed later. An introduction to Foss Electric and the specific project we studied (the S4000) project is given.

2.2.1. Foss Electric

The case study[1] was conducted at Foss Electric. Foss Electric develop and produce highly specialized and complex instruments for analytical measuring quality parameters of agricultural products. Foss Electric is among the largest manufacturers of such products in the world and they hold a large share of the world market. The company is localized in Denmark with service and distribution offices spread all over the world. The Foss Electric holding company employs approximately 700 people. The customers are laboratories, slaughterhouses, dairies, etc.

Due to high market specialization and few competitors, the innovation towards new, better and faster measuring techniques is the most important strategic goal for the company, which in turn imply the necessity of research and development.

Most of the instruments Foss Electric manufacture are used for measuring the compositional quality of milk (e.g., fat content, protein, lactose, bacteria, etc.). Other instruments are used for measuring the composition and micro biological quality of food products. The measurement technologies are typically infrared, fluorescence microscopy, or bacteriological testing.

Foss Electric has implemented concurrent engineering (Harrington, 1984; Helander and Nagamachi, 1992) yielding integration between manufacturing functions throughout the development process. Hence, the organization is very much structured in terms of projects. These projects typically include specialists with competence in fine mechanical-, electronical- and software design. In some projects also specialists in optics and chemistry are involved. Our field study con-

1 The case study is thoroughly analyzed and described as in Borstrøm et al. (1994), Carstensen (1994), Carstensen and Sørensen (1994), Sørensen (1994a), Sørensen (1994b) and in Sørensen (1994c). The description given in this section is mainly based on these analyses.

centrated on one of the large projects recently accomplished by Foss Electric. The development of the System 4000 (S4000).

2.2.2. The S4000 project

The objective of the S4000 project was to build a new instrument for analytical testing of raw milk. It was the first instrument including functionality that previously had been placed in several instruments. Further the S4000 system introduced measurement of new parameters in the milk (e.g., urea and critic acid), and the measurement speed was radically improved. The instrument consists of approximately 8000 components grouped into a number of functional units, such as: Cabinet, pipette unit, conveyer, PC, other hardware, flow-system, and measurement unit. The instrument is illustrated in Figure 2-1 below.

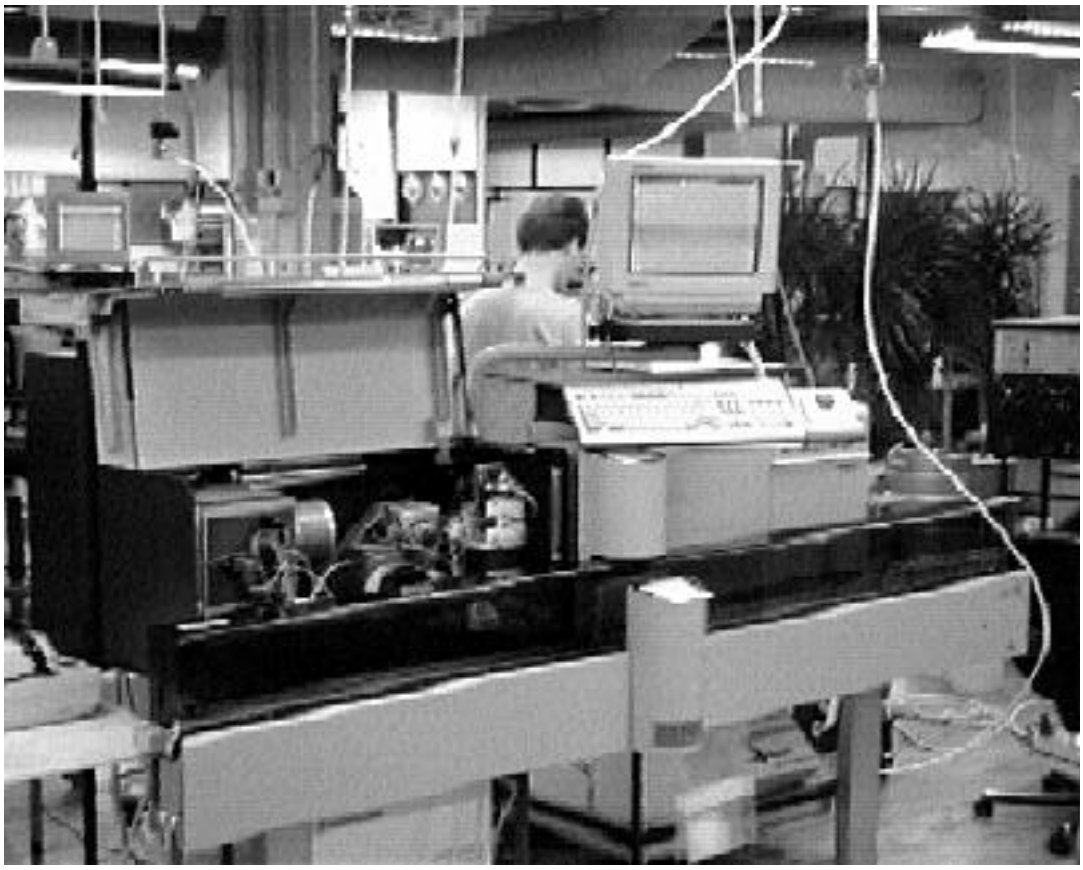


Figure 2-1: The S4000 system being tested in the Quality Control department.

More than 50 people have been involved in the S4000 project, which lasted approximately 2 1/2 year (for version 1 of the instrument). The core personnel, involved in the design included a number of designers from each of the areas of mechanical design, electronical design, software design, and chemistry. Added to this was a handful of draught-persons and several persons from each of the departments of production, the model shop, marketing, quality assurance, quality

control, service, and top management. The group of software designers that designed and coded the software to operate and control the S4000 was a sub group of the total project team having between 5 and 10 members during the development.

The S4000 is the first product with an Intel-based 486 PC build-in. Configuration and operation of the instrument is done via a Windows user interface, i.e., the user-instrument interaction is based on a graphical user interface and use of mouse and keyboard. Version 1 of the software contained approximately 200.000 lines of source code. Apart from the software designers, people from three other departments were involved in testing the software. These were the departments of service, marketing, quality assurance and quality control.

One of the central activities when developing the software is, of course, testing the software. During the S4000 project, the software designers realized problems in coordinating, controlling, monitoring, and handling the testing activities. A standardized bug form was invented and used for registering identified bugs. A set of procedures and conventions for the use of the form was established too. Some of these were written down as organizational procedures. Others were conventions that were developed during the use of the form. The purpose of the form and the procedures was to support a decentralized registration of bugs, support a centralized handling and decision making on how to overcome the identified problem, and to support that the correction activities can be conducted in a decentralized manner. Furthermore, the collection of bug forms was intended to provide an overview of the state of affairs of software testing. We have earlier defined the bug form and the related procedures and conventions as a social mechanism of interaction called “the bug handling mechanism” (cf. Carstensen, 1994).

2.3. Overall requirement for computer support for cooperative software testing

Section 5 will establish a set of detailed requirements for a computer based mechanism supporting the articulation aspects of handling the bugs. The articulation of software testing contains, however, other aspects than handling the registered bugs. For example, organizing the test in a way that ensures all modules are thoroughly tested, or the negotiation of a useful set of acceptance criteria for the software. This section will thus describe a set of overall requirements for computer support of the articulation of cooperative software testing without explicitly addressing the handling of bugs.

We have previously analyzed the complexity of the articulation of the software testing at Foss Electric and discussed how computer support of these activities could be provided (cf. Carstensen et al., 1994). The descriptions of the overall requirements in this section are mainly based on this analysis.

First, we provide a short introduction to the dimensions of complexity in articulating software testing and the dimensions of objects along which the

articulation is conducted. After this we discuss a set of overall requirements for computer support of articulating software testing activities.

2.3.1. The complexity of the articulation of software testing

Due to the fact that an exhaustive test of software is impossible (Myers, 1979; Parnas, 1985), and the different perspectives among the actors involved in the testing work, articulation work is required in order to: Negotiate the conceptual understanding of the software system, negotiate the use of bug classifications, handle the registered bugs and their correction, etc. Further, communication about software and testing is difficult because it is difficult, at a glance, to determine the state of affairs in the field of work. The work is complicated by the fact that the state of affairs is hidden in abstract representations (Parnas, 1985).

The articulation of the software testing is characterized by several factors. The field of work is undergoing constant change, making it very difficult to be aware of the state of affairs. There are interaction and interdependencies between the software components. The conditions (e.g., the software acceptance criteria) under which the work must be conducted is unstable, and the software testing activities require involvement of many mutually interdependent actors. The articulation of the software testing at Foss Electric was mainly accomplished through meetings (both ad hoc and structured) and by use of different forms, lists, etc. (cf. Borstrøm et al., 1994).

The articulation of the process of testing the software in the S4000 project included activities such as: Classification of bugs and structures, allocation of resources, planning and scheduling of tasks, incorporation (or mesh) of tasks, coordination of ongoing activities, negotiation of classifications, allocations, criteria, priorities, etc., monitoring the state of affairs in the development and test process, and distributing information. The activities were mainly based upon conceptualization reflecting structures from the field of work (the software being tested) or from the work arrangement, aggregation of detailed information, and classifications and categorizations. The most common used conceptualizations reflected: The software architecture, i.e., the modules and their interaction, aggregations of bugs with respect to category, priority, module, etc., available human resources addressing especially their capabilities and work load, and existing plans, tasks, deadlines, etc. and their interrelations. Figure 2-2 illustrates the central conceptualizations used for articulating the software testing and the basic activities involved in the articulation work. These are more thorough described in Carstensen et al. (1994).

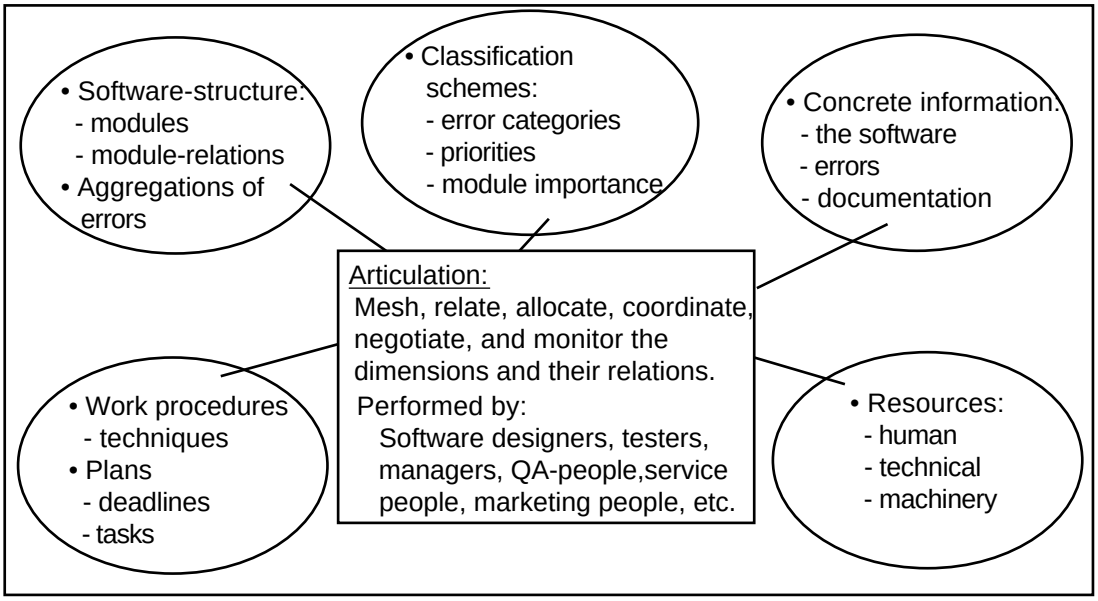


Figure 2-2: A graphical model of the articulation of the software testing process at Foss Electric. The circles are the dimensions (the conceptual structures) along which the work is articulated. The box contain the main functions to be fulfilled. The structures in the upper circles are mainly derived from the nature of the field of work (software testing). The structures in the circles at the bottom mainly reflects the nature of the cooperative work arrangement implemented at Foss Electric. Neither the list of the conceptual structures, nor the list of basic functions should be viewed as an exhaustive lists. Adapted from Carstensen et al. (1994).

A first obvious overall requirement for computer support of the articulation of software testing is that the listed conceptual structures and information collections are accessible and modifiable, and that the activities listed is supported. If we consider a computer system as consisting of a set of data structures and a set functions, the data structures should be symbolic representations of the above mentioned conceptual structures, and the functions must reflect the activities mentioned as accessible operations on the conceptual structures. The conceptual structures can be seen as the dimensions of objects of articulation (see Schmidt et al., 1993), i.e., as the dimensions along which the articulation work is organized. The table listed below (Figure 2-3) summarize the dimensions of objects of articulation addressed by the bug form mechanism.

Symbolic reference	Symbolic reference in mechanism (the form, the binder, and the procedures)	Functions in the mechanism
Objects of articulation work		
Actors	Tester Responsible designer (Spec.-team) (Central file manager) (Platform master)	Assign
Responsibilities	The form is a responsibility relation between the responsible designer and the correction task identified. (Consider consequences) (Validating corrections)	Allocate, Hand over, Accept, Reject, Negotiate
Tasks	The correction of the bug (the content of the form itself) (Consider consequences) (Validating corrections)	Accept, Reject, Accomplish, Negotiate
Activities		
Conceptual structures	Classification of bugs Modulation of the software Working rhythm (platform periods) Stages for a correction process	Define, Specify, Classify, Relate, Direct attention to, Negotiate,
Technical resources	Test machine	Specify
Informational resources		
Material resources		
Infrastructural resources		
External systems of reference		
Work environment		
Field of work	Classification of bugs Software structure (modules) Files	See conceptual structures above
Organizational setting		
Space		
Time	Platform period Registration time Diagnosis time Correction time Testing time	Specify, Relate to, Negotiate, Refer to

Figure 2-3: The objects of articulation handled by the bug form mechanism and the functions related to these. Adapted from Carstensen (1994).

2.3.2. Requirements for facilities provided by a computer system

This section presents the essential requirements for computer support of the articulation of software testing, as identified at Foss Electric. It should not be regarded as an exhaustive or prioritized set. Several of the requirement mentioned in the following has been included although they are not related to the bug handling

mechanism. Limitations regarding what should be required by the “new” computer based bug handling mechanism will be discussed in the beginning of the next section.

Some requirements mainly concern the data structures that must be provided, whereas others concern the functionality. A general requirement is that all data structures or relations between data structures mentioned must be available to the actors, and in most cases they must be modifiable too.

A computer system supporting the articulation of complex software testing processes must provide access to data structures reflecting the architecture of the software complex. Main functionality, relevant classifications, and the relations to other modules must be included for all modules. Furthermore, references to the responsible software designers, relevant documentation, and specification must be available. All the structures must be modifiable.

Access to descriptions of the specific bugs registered must be provided. The descriptions must contain information on originator, symptoms, priority, suggested diagnosis, involved modules, estimations, related responsibilities, correction, etc. The information must be accessible as aggregations with respect to modules, types of bugs, priorities, and responsible designers too.

A third type of conceptual structures that must be available is existing plans containing information on the use of both human and technical resources. The relations among tasks, deadlines, actors, software modules, etc. must be accessible. This includes access to a conceptual structure containing information on all involved actors, and to technical and hardware resources involved in the development and testing. Individual characteristics, present and planned workload, etc. of the actors and technical resources must be available.

A central aspect of articulation work is classifications of the structures, objects, situations, etc. from the field of work or the work arrangement. This is also a central aspect of articulation of software testing. This implies the need for access to data structures containing information and descriptions of the present classification categories for bugs, software modules, etc. The classification categories must be accessible in all work situations where they are used (e.g., in error registrations). Furthermore, support of distributed changes to the classification schemes must be supported. This will require some kind of structured communication channel and/or a mechanism structuring the negotiation process.

The last conceptual structure concerns the organizational context. A computer support system must also provide access to structures of information on organizational procedures, techniques, standards, etc. and to structures containing information on the requirements for the software complex (both the existing specifications and the suggested interpretation) used in the test work.

The first obvious requirement for functional support addresses distributed registration and classification of bugs, and support to the testers in filling in all the information required in order to perform the diagnosis. Furthermore, the access to specify (and later on modify) a work flow process stipulating to whom informa-

tion on the bug must be routed, how the bug registration is made visible to other testers, etc. must be offered.

The articulation work is often handled in a distributed manner. Facilities must be provided for a distributed creation and registration of new tasks. Tasks must be established as relations between a task, an actor, and a deadline. Functions for integrating the tasks in the existing plans and for notifying the affected actors are required. The structure for how, and whom, to notify must be available as a modifiable structure describing how this stipulation should run. This requires, of course, a communication structure allowing the actors to send requests, accepts, and rejects to each other.

The requirement in the previous paragraph naturally leads to a requirement for a communication channel where the actors can negotiate a diagnosis, the resource allocations, a deadline, etc. by use of structured messages and a predefined message flows. The structure of the messages and dialogue flow must be modifiable, i.e., tailored to the concrete situation.

An important aspect of articulation work is monitoring the state of affairs. Thus features are required that supports the testers and designers in making it possible for others to be aware of registered bugs and implemented corrections. Both user activated and automatic distribution of this type of information must be available. And all actors must, upon request, be able to receive information on the state of affairs. This can be fulfilled through access to the registered bugs, their diagnosis, and status, or through access to information on a specific correction task.

The listed requirement should not be regarded as an exhaustive list. Rather they illustrate the recommendations we established on the basis of a first analysis of the complexity of the articulation of software testing at Foss Electric (see Carstensen et al., 1994).

2.4. Handling software bugs in the S4000 project

This section describes the analyzed paper based mechanism[1] handling software bugs at Foss Electric.

The purpose of the bug handling mechanism is to support and stipulate the process of 1) registering a bug, 2) diagnosing the bug, 3) estimating the correction time, 4) correcting the bug, and 5) verifying the correction. The following subsections characterize the actors involved in the process (section 4.1), present and discuss the form used for handling and mediating the information (section 4.2), and finally describe the process as a stipulated flow and of the ongoing activities (section 4.3 and section 4.4).

---

1 The bug handling mechanism is thoroughly analysed and described in Carstensen (1994). The description below is mainly based on this analysis.

2.4.1. The actors involved in software testing

Several groups of actors are involved in the process of registering, diagnosing, correcting, and verifying. They are thus users of the form and the procedures. The relevant groups of actors are:

(1) The testers.

These are the actors involved in the actual testing of the software in the S4000 instrument. They come from several different departments and they have very different background and approach to what the software must provide. Typically the testers are software designers, people from quality assurance and quality control, electronic designers, marketing people, people from the service and maintenance department, etc.

(2) The spec.-team.

This is a group of three software designers being responsible for diagnosing the bugs and deciding how to handle the correction of the bugs. The members represent different areas of expertise in the software architecture, i.e., one has deep knowledge on aspects regarding the user interface and the used file structures, etc., one has deep knowledge of the algorithms used for computing the measuring results, and one is very experienced in developing software interfacing the network, the hardware, and the external devices controlled by the software.

(3) The software designers.

All software designers are responsible for one or more software module. Since a bug is always related to a specific module, one of the designers will always be responsible for correcting a specific bug.

(4) The central file manager.

All registered bugs are filed in a binder accessible to all testers and designers. One of the software designers is responsible for organizing and maintaining the central file organized in a binder. He is also responsible for informing the platform master about which bugs must be verified in the next integration period.

(5) The platform master

The platform master is one of the designers in the project. He is responsible for managing and coordinating the activities in one of the integration periods (for a description of this cf. Borstrøm et al., 1994). He is, amongst others, responsible for verifying the corrections made by the designers.

The following describes how these actors, more precisely, are involved in the process and how the information and requests are circulated among them during the test and correction work.

2.4.2. The bug form

The bug form was invented and refined during the early stages of the S4000 project. The overall purpose was to establish a mechanism that could ensure that all identified bugs were registered and “remembered” until they were corrected. A procedure prescribes that only one form per registered bug must exist. The state of the form illustrates the state of the bug and its correction. To ensure that all bugs are handled, the central file (the binder) contains a copy of the form until a final state is reached, and the original form can be filed. The content of the bug form is illustrated in Figure 2-4 below.

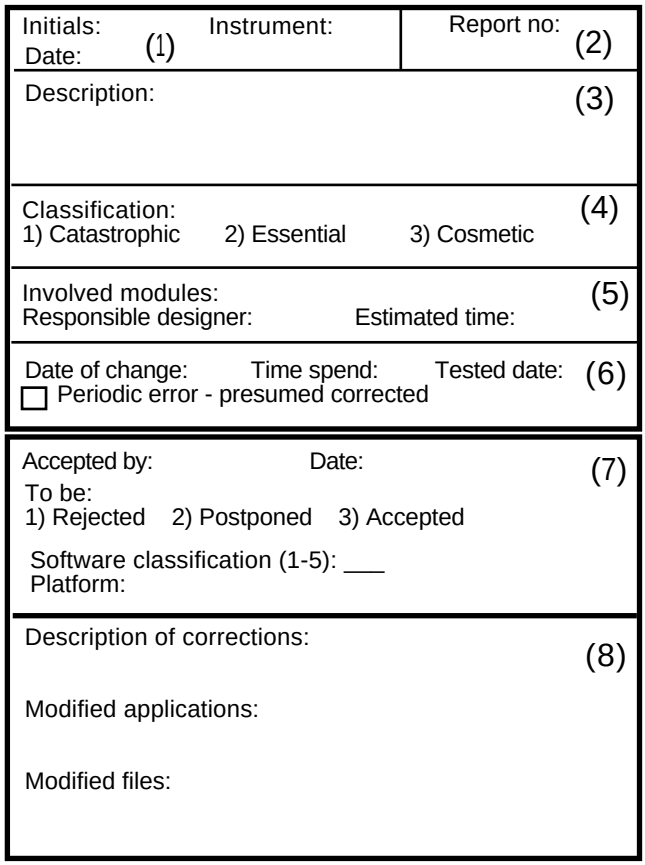


Figure 2-4: The bug form used at Foss Electric. The form is a sheet of A4 paper printed on both sides. The upper square represents the front, and the lower square represents the back of the form. The numbers in brackets indicates which actors fill in the information in the form. The described procedure illustrates the overall registering, diagnosing, correcting, and verifying flow.

All registered bugs are filed in a ring binder. This is accessible to all software designers and other testers. It is physically placed in the same room as the project team.

The purpose of the binder is to give all involved designers and testers an opportunity to be aware of the state of affairs in the testing of the software, and to

establish an overview of the progress of the testing process, e.g., be able to see the number of bugs not yet corrected.

The binder has seven entries and in each of these entries the forms are filed in a chronological order. The seven entries are: 1) Non-corrected catastrophic bugs (copies), 2) Non-corrected essential bugs (copies), 3) Non-corrected cosmetic bugs (copies), 4) Postponed bugs (originals), 5) Rejected bugs (originals), 6) Corrected bugs not yet verified (copies), and 7) Corrected bugs (originals).

These entries reflects the status of a specific bug, and they play a central role in stipulating the articulation of some of the activities involved in testing the software.

2.4.3. The process of handling bugs

We will use three different approaches to get an overview of the process of registering, diagnosing, correcting, and verifying bugs: Illustrate the bug handling mechanism as a state-transition diagram, go through the twelve steps in the process mentioned in Figure 2-4 and describe each step a bit further, and address the process from the point of view of the involved actors.

The state-transition model illustrated in Figure 2-5 below reflects the possible different states of the bug handling mechanism. The transitions going out from a state illustrate the next possible and “legal” steps. It can, thus, be viewed as a model of the mechanism handling the articulation. It is not a model of the states, procedural steps, or of the activities involved in the concrete testing work itself. Neither is it a model reflecting the different actors involved in work.

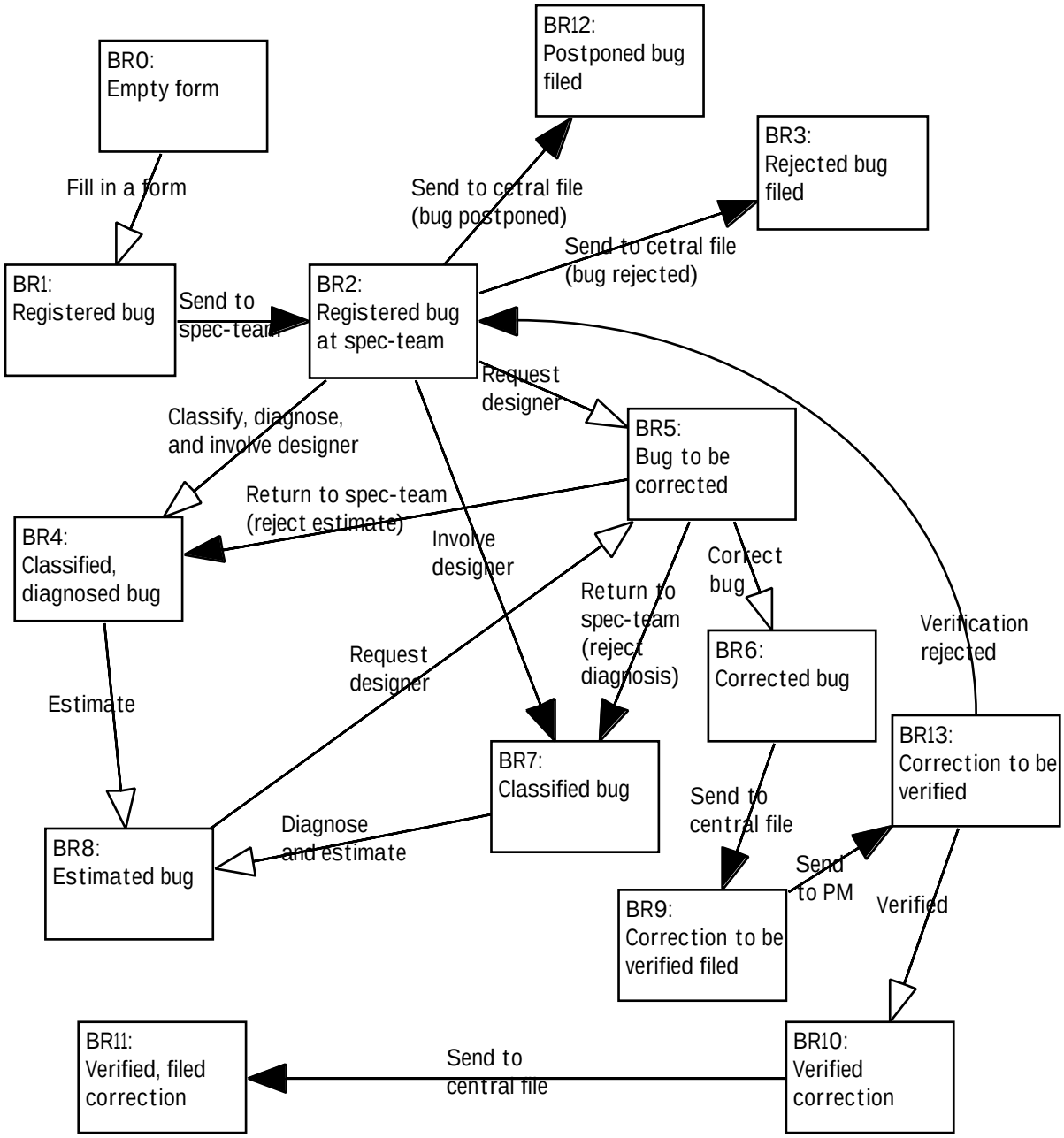


Figure 2-5: A state-transition diagram for the mechanism used for supporting the articulation of the software testing work in the S4000 project. Each box illustrates possible states of the mechanism. The arrows illustrate possible actions (transitions) that can be taken and points at the new state a given action will result in. Arrows with a white arrow head indicate changes to the content of the mechanism; here mainly in terms of adding new information to the form. The arrows with black heads indicate changes in who is in control of (allowed to change the state of) the mechanism.

The following description of the general procedure to be followed will relate the procedural steps to the states and transitions illustrated in Figure 2-5.

The state-transition diagram can be read as an illustration of the process of registering, diagnosing, correcting, and verifying bugs and their correction. This

process, without any exceptions or need for involving extra expertise, is illustrated if we follow the "main road" of: BR0 → BR1 → BR2 → BR5 → BR6 → BR9 → BR13 → BR10 → BR11.

When analyzing a social mechanism of interaction (here the bug handling mechanism) a central approach is to understand and describe what triggers the different transitions in the state-transition diagram. This illustrates some of the conditions that affects the articulation work. For the purpose of this section the triggering conditions will, however, not be explicitly addressed. A discussion of these can be found in Carstensen (1994).

The overall procedure for how the bug handling mechanisms works and the how the forms and central file are used and updated consists of twelve overall procedural steps:

(1) A bug is recognized by a tester. The bug is classified and a form is filled in by the tester.
(2) The form is send to the spec.-team.
(3) The bug is re-classified and diagnosed by the spec.-team.
(4) The responsible module and designer are identified by the spec.-team.
(5) The correction time is estimated by the spec.-team.
(6) The required correction tasks are incorporated in the work plans.
(7) The responsible designer is requested to correct the bug and a copy of the form is send to the central file (the binder).
(8) The bug is corrected and additional correction information is added on the form by the responsible designer.
(9) The designer sends the form to the central file.
(10) The central file manager sends the form the platform master and inserts a copy of the form in the binder.
(11) The correction is verified by the platform master.
(12) The accepted forms are returned to the central file.

We will describe the procedure further in the following. In most cases the prescribed procedure will be followed quite strictly, but there are, of course, situations where the actors choose not to follow the procedure. Exceptions will not be detailed discussed here. The "typical" exceptions are described in Carstensen (1994).

First step is reporting a bug. When a tester identifies a bug a form is filled in. The tester fills in his or her initials, the date, an identification of the instrument and the software version he used, and a description of the input and output in the situation when the error occurred. Finally the bug is classified as either catastrophic, essential or cosmetic according to the assessed importance of the problem. These actions are reflected in Figure 2-5 as the transition from state BR0 to BR1.

The next step is that the tester sends the bug form to the spec.-team (the BR1→BR2 transition). Usually this is done by use of the internal mail, but the

tester might decide that the diagnosis and correction cannot be delayed further. If this is the case he or she might go directly to one of the spec.-team members (or another designer) in order to discuss it immediately.

In “normal” work periods the spec.-team typically meets ones a week. Each of the received bug forms are discussed. First, the spec.-team decides if the described bug can be accepted as a bug. If not, the form is classified as rejected and filed in the “Rejected bugs” entry in the binder (the BR2 → BR3 transition). This is also the case if the decision is to postpone the bug, except that the “Postponed bugs” entry is used (BR2 → BR12).

The remaining forms are classified as accepted and is then classified according to it’s importance. First, the classification made by the tester is assessed. If the spec.-team disagrees with the classification they change it. If it is considered necessary, the spec.-team might choose to contact the tester and negotiate the classification. The development is organized working rhythms called platform periods (cf. Borstrøm et al., 1994) each on 4–6 weeks. This implies that the platform period in which the bug must be corrected is decided and following documented in the form.

If the diagnosis of the problem and the estimation of time required to correct the bug is fairly simple, the spec.-team fills in information on the diagnosis, the responsible module(s), and the responsible designer(s) in the form. Each module is assigned a responsible designer. This implies that diagnosing which modules are affected by the bugs leads to assigning the designers responsible for correcting the bug. Then the spec.-team incorporate the correction task(s) in the plans and sends a request (the form) to the designer (BR2 → BR5).

If the diagnosis is complicated the spec.-team can choose to call in the designer(s) responsible for the relevant modules (BR2 → BR7). The designer(s) is then involved in diagnosing the problem and estimating the correction time (BR7 → BR8). Hereafter the spec.-team incorporate the correction task(s) in the plans and sends a request (the form) to the designer (BR8 → BR5). In some cases the diagnosis is simple but the estimation is complex. In these situations the spec.-team and the responsible designer negotiate the required corrections time (BR4 → BR8). As for the previous situations, the spec.-team then incorporate the correction task(s) in the plans and send a request (the form) to the designer (BR8 → BR5).

In order to incorporate the correction tasks into the plans, the spec.-team inform the designer or manager responsible for the overall project plan to include the new tasks in the plans. He will then add the tasks, responsibilities, deadlines, and estimations to the plans. In the S4000 project the person responsible for the overall plan was one of the spec.-team members. Handling the plans is not considered part of the bug handling mechanism. Thus, no transitions are triggered. Instead it is an example of a link between two mechanisms supporting the articulation of software testing and development: One handling bug registration and one handling scheduling and allocation of tasks and human resources.

After the diagnosis and estimation the bug, the form is send to the responsible designer, i.e., the designer is requested to correct the bug (BR2→BR5 or BR8→BR5). A copy of the form is filed in the binder in the relevant entry (“Non-corrected catastrophic”, “Non-corrected essential”, or “Non-corrected cosmetic”).

The designer assesses the diagnosis, the estimate, and the deadline (the platform period). If he consider the estimate too low, he returns the form with a note stating that the estimate is unacceptable (BR5→BR4). The estimate will then be negotiated with the spec.-team. If the diagnosis is considered wrong or not acceptable, a similar procedure is used (BR5→BR7), and the diagnosis and the estimate is negotiated with the spec.-team. If the designer can accept the diagnosis he corrects the bug. This might, of course, be at a much later point in time. Identifying the source of a bug and eliminate it is a complicated task that requires intense studies of the source code and specifications, discussion with other designers, etc. This task is not addressed in this chapter.

Having corrected the bug, the designer fills in the relevant correction information in the form (cf. Figure 2-4). This is reflected as the BR5→BR6 transition in Figure 2-5. Next the designer sends the form to the central file manager. The central file manager then removes the old copy of the form from the binder and insert a new copy of the form (containing the additional information on the corrections) in the entry on “Corrected bugs to be verified”(BR6→BR13).

A few days before the next platform integration period the central file manager sends the original form to the platform master (BR13→BR9). The platform master is responsible for verifying the corrected bugs, i.e., control that each bug is corrected sufficiently without introducing new problems. The verification can be done either by the platform master or by delegating the responsibility for doing this to one of the software designers. If the verification process results in an acceptance of the correction the form is categorized as accepted (BR9→BR10). If the verification process results in a rejection the platform master adds a note in the form indicating the problem and sends it to the spec.-team (BR9→BR2). If the correction of the bug is considered verified, but has introduced one or more new bugs, the form is classified as verified (BR9→BR10) and a new empty form is filled in describing the new bug. This starts a completely new process (BR0→BR1).

Finally, the forms classified as verified are send from the platform master to the central file manager. The central file manager removes all the form copies in the “Corrected bugs not yet verified” entry of the binder, and insert the received originals in the “Corrected bugs” entry (BR10→BR11).

2.4.4. The process seen from the actors perspective

As described earlier, several actors having several different roles, are involved in the process of handling the process of registration, diagnosing, correcting, verifying software bugs and their correction. If we approach this from the perspective of the involved actors the process can be illustrated as shown in Figure 2-6 below.

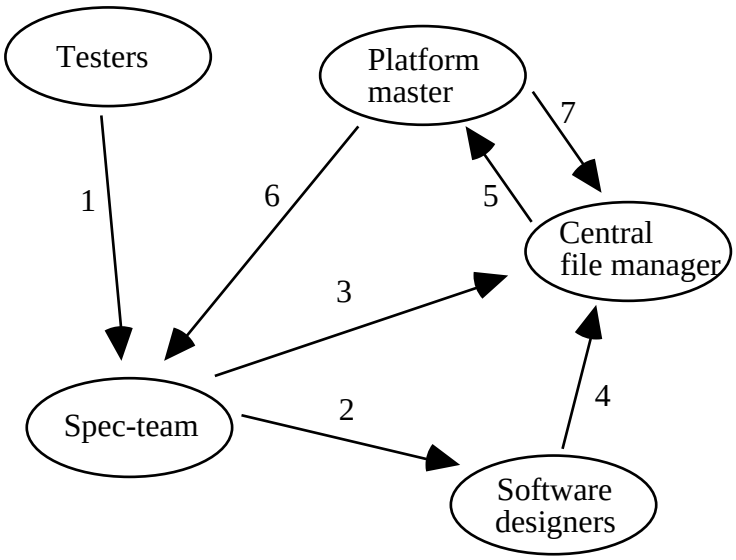


Figure 2-6: A visualization of the actors (roles) involved in testing the software of the S4000 instrument and the information flow between them. The information flow described in the figure illustrates the intended flow of the bug handling mechanism. The flow (the route of the forms) follows seven major steps: (1) A tester send a form describing a recognized bug to the spec.-team. (2) The spec.-team adds diagnosis and estimation information and sends it to a software designer. (3) A copy is send to the central file manager. If a bug is rejected the original is send to the central file manager. (4) The software designers add correction information to the form and sends it to the central file manager. (5) The central file manager sends a pile of forms to be verified to the platform master. (6) Forms that cannot be verified are send to the spec.-team. (7) Verified forms are send to the central file manager. The figure is a simplification of figure 3-3 in Carstensen (1994).

This chapter aims at establishing a set of requirements for computer support of the articulation aspects of software testing. When discussing such requirements, several approaches to understanding the work must be taken. Some of these have been illustrated in the previous: A function oriented approach addressing what the mechanism provides to the actors, and how this is reflected in terms of the possible states of the mechanism. A process oriented approach describing what happens during the flow of the registering, diagnosing, correcting, and verifying process. And an approach reflecting how the flow is seen from the perspective of the individual actors. Before establishing requirements other approaches might be taken in order to get an even more coherent understanding of the work setting to be supported. Such approaches could address the constraints and conditions under which the testing work must be undertaken, characteristics of the organizational context, characteristics of the power structures in the work setting, the educational background and competence of the actors, etc. Since the purpose of the requirements discussed in this chapter is mainly illustrative, we have, however, decided to stop the description here and continue with the requirements.

2.5. Requirements for a computer based bug handling mechanism

In this section we aim at presenting and discussing the requirements for a computer based mechanism supporting articulation of distributed software testing activities, i.e., distributed registration of bugs, routing the required information to the actors, and stipulate the sequence of activities and the involvement of actors in the register-diagnose-correct-verify process. In the rest of this chapter the computer based bug handling mechanism will be referred to as the "Bug-MOI" (computer based Bug handling Mechanism of Interaction).

The discussion has been organized as a top-down process, i.e., starting from general requirements (presented in section 3) towards more concrete requirements and sketches of how the Bug-MOI could be implemented. Distinguishing in any exact manner between requirement specification and systems design is widely recognized as an impossible task (Andersen et al., 1990). We have, however, as a pragmatic approach to specifying the Bug-MOI tried to distinguish between, on the one hand, which functions it must perform on which data, and on the other hand, how these functions are performed and what the Bug-MOI should look like.

Section 5.1 takes a first step in delimiting the set of overall requirements we will address. The intention of this chapter is to illustrate how concrete requirements for a Bug-MOI could be specified in order to provide input for a refinement for concept of mechanisms of interaction and the architecture and notations related to this (cf. chapter 3 in this book). Thus, we will not specify a complete set of requirements. Section 5.2 takes the next step by discussing the basic functionality of the Bug-MOI and the interactions between the actors and the Bug-MOI. Based on this, Section 5.3 describes which structures the Bug-MOI must contain or provide access to. Section 5.4 presents the functions accessing the data structures, and Section 5.5 defines the links from the Bug-MOI to "external" mechanisms. Section 5.6 sketches the concrete data structures, classification types, user interface structures, etc. that must be included in the Bug-MOI, and Section 5.7 discusses how the related protocol could be designed in order to fulfill the requirements.

It is important to notice that this chapter only discuss requirements for a Bug-MOI supporting certain aspects of articulation of software testing. If the aim of this exercise had been to develop a real life system we would, of course, need to include requirements derived from other aspects of the articulation work, as well as relating the support of articulation work to requirements for support of the actual testing activities.

2.5.1. General requirements for a computer based bug handling mechanism

The requirements established in Section 3 will be used as inspiration and criteria of relevance for the discussion of the requirements for the Bug-MOI in this sec-

tion. For each of the listed requirements we have considered the relevance of including it as a requirement for the Bug-MOI. Aspects which are clearly outside the bug form mechanism are excluded in order to delimit our focus. For example, the requirements on providing access for the actors to revise, update, and get an overview of the work plans are excluded in the following discussions.

Before establishing the set of requirements, we have to make some general decisions on the allocation of functionality between the involved actors and the Bug-MOI, i.e., decide what is to be handled by humans and what is to be handled by the computer system. An example is the registration of a bug. It is the actor who decides the classification of a bug and enters all relevant information into the system, but it is the Bug-MOI that validates that all mandatory information is entered, as well as it passes on the registration to the actor(s) responsible for the next sub task to be conducted.

The Bug-MOI must *handle all information*—and aggregations of information—related to the registration of a bug, its diagnosis and correction, and its verification. Furthermore, all *routing of information* between the involved actors (or rather roles) must be handled by the Bug-MOI. The Bug-MOI is *not making any decisions* regarding whether or not a phenomenon is a bug, how to classify a bug, how to diagnose a bug, which modules a correction will affect, what a correction time estimate should be, how to correct a bug, etc. This is taken care of by the human actors. To phrase it differently: The Bug-MOI should not include any “knowledge” used when decisions are taken in the actual work, e.g., decisions on whether new expertise is required, or decisions on how to correct a bug. The Bug-MOI will only contain “knowledge” of how the work-flow must be stipulated.

Basically the Bug-MOI must provide support for three different problems: Firstly, ensure that all registered bugs are treated until they reach a final state. Secondly, ensure that the state of affairs overview provided is up to date and correct. And third handle a process of distribution, compilation, distribution, and compilation. Distribution of the registration tasks, followed by compilation of all the bug forms for diagnosing, followed by distribution of the correction tasks, and finally compilation of corrected bugs for verification. These three overall requirements result in requirements that actually limit the degrees of freedom for the actors, but this is necessary in order fulfill all three needs.

The decisions on the division of Bug-MOI—Human actor functionality implies that there basically the Bug-MOI must provide four types of functionality:

First, the mechanism must offer facilities for registering new bugs. This must be organized so that the registration can be done in a distributed manner from as many work places as we need. The Bug-MOI must support the registration so it is ensured that all mandatory information is entered before the registration is completed and passed on. All registered bugs must be filed so aggregations and statistical information on the complete set (or subsets) of the bugs can be generated.

Second, the Bug-MOI must stipulate the work flow by routing the information between the actors. When a certain actor has completed his or her activities in relation to the handling of a specific bug, the mechanism must automatically validate that the required information is registered and then pass the information on to the next actor (or group of actors), and notify the receiver(s) to indicate, that the specific bug is now at a stage where a new action must be taken. Many studies have indicated that for most work situations is it impossible to make a complete specification covering all situations that can occur (see e.g., Schmidt, 1993; Suchman, 1987). Software testing at Foss Electric is certainly no exception to this (cf. Carstensen, 1994; Carstensen et al., 1994). Thus, the actor completing an activity must be able to overrule the routing and redirect the information to whoever he or she wants. Another important requirement is that the protocol stipulating the routing must be based on roles to which actors can be related. The study at Foss Electric clearly illustrated that all actors had several roles, and, more importantly, the roles were relatively stable entities in the work setting. Many of the roles were handled by different actors at different points in time. Thus it must be possible to change the actual actor related to a role without changing the protocol stipulating the routing. And finally, receiving a notification can be regarded as receiving a request. The Bug-MOI, hence, must provide a facility that makes it possible to reject a request, i.e., return the request to the originator with a note indicating why this is rejected.

Third, the Bug-MOI must support the resource allocation tasks in relation to the diagnosis and estimation tasks. When the spec.-team decides on the diagnosis of a bug and on who is going to correct a specific bug, they also handle resource allocation (cf. section 4). To be able to do this the mechanism must provide information to the actors (the spec.-team) on the relations between roles and actors, the architecture of the software complex, the relations between software modules and the responsible designer, the workload of the involved designers, the existing work plans, and the relations between tasks and deadlines, etc. Furthermore, supporting the diagnosis task—and thus the resource allocation task—requires an improvement of the existing classifications of the importance of bugs. The existing categories are insufficient and are used for several different purposes, e.g., both as an indication of the problem from the perspective of the testers and as an indication of the importance from a software development perspective). The Bug-MOI is required to provide a more sufficient and elaborated set of categories of bugs.

Fourth, the Bug-MOI is required to support the actors in obtaining awareness of the state of affairs regarding the registered bugs. An important aspect of the original bug handling mechanism (cf. section 4) was to provide the actors with aggregated information on the number of reported bugs not yet corrected, the number of bugs with a certain classification, the number of not yet corrected bugs in a specific module, the number of bugs to be corrected by a specific designer, etc. The Bug-MOI must provide a series of querying facilities for generating such aggregations, including support for getting detailed information on a specific bug.

It should be noted that for articulation purposes, is it important to be able to monitor the progress of testing activities and the state of affairs in general, i.e., to get an overview of how many percentage of the code is tested, what is the number of man-hours planned for testing, how much time has been spend on testing so far, etc. This type of information should certainly be provided to the actors by a computer based mechanism, but this requirement is considered beyond the scope of this chapter.

The requirements discussed so far contain no reflections on whether all the facilities should be provided directly by the Bug-MOI itself, or whether they can be provided through links to other mechanisms. It seems obvious to expect that some of the facilities are provided by the Bug-MOI, which then is facilitated by other computer based mechanisms. For example, information on the software architecture could with benefit be obtained in a data-structure maintained by another computer based mechanism such as a programming environment or a CASE tool.

In order to reduce the complexity of the task of specifying the requirements for the Bug-MOI, we have *excluded* supporting certain aspects of the articulation of software testing, of which the most essential are:

(1) Negotiation structures.

A very important aspect of articulation of software testing is the negotiation on classifications, diagnosis, resource allocations, deadlines, etc. It would be obvious to require the Bug-MOI to provide some predefined structures that could be used by the actors when negotiating, for example, a diagnosis or an estimate, or when negotiating, for example, changes to existing classification categories, acceptance criteria, work procedures, etc. We have in the present chapter decided not to discuss such predefined structures. We will however, still require, that all requests send from one actor to another can be rejected and routed back to the originator. A more elaborate support for actors engaging in negotiating the software testing process could draw upon research addressing the support of negotiation, such as Flores et al. (1988) or Simone et al. (1994).

(2) Decision support.

Apart from providing relevant information, the Bug-MOI should not provide any decision support facilities. As mentioned, we have chosen to let all decisions on, for example, the classification of a bug, the diagnosis, how to correct a bug, etc. be taken by the actors without any active intervention from the Bug-MOI. We consider it as far beyond the scope of this chapter to discuss such ideas.

(3) Communication facilities.

It might be relevant to set up requirements for the Bug-MOI, for example, to support the members of the spec.-team in being geographically placed in different places, and then use the mechanism as a means for communicating on the diagnosis of a bug and on the estimate of the correction time. Such general communication facilities are

considered out of scope here. This problem, could be addressed by some of the same means as addressing negotiation (Flores et al., 1988; Simone et al., 1994), collaborative writing, video-conferencing, etc. (Ellis et al., 1991; Ishii et al., 1993).

(4) Access to updating plans.

A central aspect of articulating the handling of the bugs identified in a test is, of course, to be able to see the current plans and make changes to these. We have chosen to see the support of updating the work plans as a separate computer based mechanism. In this chapter the linking of the two mechanisms will only be addressed through a requirement stating, that information on a task generated by the accept of a bug must be routed to the actor responsible for handling the work plans. It seems obvious to require a much more advanced linking between the two mechanisms, e.g., as an automatic update of the plans combined with a notification of the actor(s) handling (responsible for) the planning. We have, however, decided to exclude this from the requirements. Further work on the requirements could involve a specification of the linking between the two computational mechanisms of interaction, amongst others, based on the analysis of how the social mechanisms of interaction are linked presented in Schmidt et al. (1994).

Neither the mentioned requirements nor the list of aspects that are excluded must be regarded as exhaustive. During the process of refining the requirements several of the requirements will be specified further, and further decisions on what to include and what to exclude are taken.

2.5.2. The interaction between the actors and the Bug-MOI

This section discusses the interaction between the actors and the computer based bug handling mechanism (the Bug-MOI). We will do this by step by step go through a description of which actions the actors (the users of the Bug-MOI) perform, and what kind of result this leads to in the Bug-MOI. The Bug-MOI must provide two general facilities. One supporting the registering-routing-diagnosing-correcting-verifying process (shown in Figure 2-7), and one facilitating browsing and search activities (shown in Figure 2-8 to 2-11).

For presentation purposes, the description of the types of interaction with the Bug-MOI illustrates the “typical” working sequence, i.e., the flow of actions as they would appear in a typical situation without any exceptions. We will thus follow the “main flow” of states and transitions (from BR0 to BR11) in the state-transition diagram in Figure 2-5.

Each row in the tables (Figure 2-7 to 2-11) is a pair of “user action—computer mechanism responds”. First column describes the user actions and the next column contains the related reaction from the Bug-MOI. In order to relate the description to the case description earlier, and to illustrate a possible future use of the Bug-MOI, we have included in the descriptions the roles the actors typically

will have. Decisions on the allocation of functionality between humans and computational artifacts are, thus, described as a set of scenarios (Cambell, 1992; Karat and Karat, 1992). The registering-routing-diagnosing-correcting-verifying process is presented in Figure 2-7.

Actions from the actor(s)	**The responds from the Bug-MOI**
1) A tester recognize a bug in the software, decides to report it, and “generate a new bug report”.	The Bug-MOI replies by setting up an electronic form containing entry fields for the relevant information.
2) The tester classifies the bug, fills in the fields in the form, and ask the system to “register the bug”.	The Bug-MOI validates that all mandatory fields are filled in. If not the tester is requested to do this. When all mandatory fields are filled in the registration is filed in the central bug database and a notification is send to the spec.-team.
3) A spec.-team member asks for the “next new registration”.	All registration information are presented to the spec.-team member together with relevant fields to be filled in concerning the diagnosis and estimation.
4) If the bug cannot be accepted by the spec.-team, a spec.-team member demands to “reject bug”.	The bug is filed as “rejected” in the central database, and a notification is returned to the tester indicating that the registration has been rejected.
5) If the bug is accepted, but it is decided to postpone it, a spec.-team member classifies the bug, describes the reason to postpone it, and demands to “postpone bug”.	The bug is filed as “postponed” in the central database, and a notification is returned to the tester indicating that the registration has been postponed.
6) If the bug is accepted by the spec.-team, a spec.-team member fills in the classification of the bug, the platform period in which it is going to be fixed, and the responsible module(s).	On the basis of the specified module, a default responsible designer is added to the information.
7) The responsible designer(s) are filled in together with the correction time estimate for each. “Bug accepted” is demanded.	The bug is filed as “accepted” in the central database. The responsible designer(s) and the originator (the tester) are notified.
8) A designer demands a “see correction request”.	All information on the registered bug is presented to the designer.
9) If the designer rejects to do the corrections or cannot accept the correction time estimate he fills in a rejection description and asks to “reject request”.	The bug is filed as “correction request rejected” in the central database and the spec.-team members is notified. The spec.-team can then handle it as a new registration (cf. entry 3).
10) If the designer accepts the diagnosis and estimate he demands a “accept request”.	The bug is filed as “correction request accepted” in central database and the spec.-team members are notified.
11) The designers asks for a “register correction”.	Identification information on the bug is presented to the designer together with fields for registering information on the corrections.
12) The designer fills in the time spend and information on affected modules and files, and demands a “register correction”.	The bug is filed as “corrected” in the central database.
13) The platform master asks to “see corrections to be verified”.	Information on all corrections to be verified in the next platform integration period is presented to the platform master.
14) The platform master demands a “register verifications”.	Information on the next bug to be verified is presented to the platform master.
15) If the bug presented cannot be verified the platform master fills in a description of the problem and demands “verification rejected”.	The bug is filed as “not corrected” in the central database, and the spec.-team members are notified. The spec.-team can handle it as a new registration (cf. entry 3).
16) If the bug presented can be verified the platform master demands “verification accepted”.	The bug is filed as “corrected and verified” in the central database.

Figure 2-7: A table of the typical user actions and Bug-MOI reactions (each pair in the rows) in the registering-routing-diagnosing-correcting-verifying process.

The description of the “typical flow” of the registration and correction process illustrated above does not contain examples of exceptions where, for example, the actor choose to overrule the stipulated workflow and skip some of the steps in the process. It should, according to the overall requirements discussed in section 5.1, always be possible for the actor to do so.

As mentioned above, support for browsing and search is the second general facility provided by the Bug-MOI. Although this does not provide a similar “natural sequence” as in the previous facility, we use the same scenario structure in order to describe the browsing and search facility. This is done by presenting four prototypical situations: 1) An actor (usually a tester or a designer) is interested in obtaining specific information on a specific bug (shown in Figure 2-8); 2) A tester who wants to check if a bug registration identical to a phenomenon he has just recognized already exists (shown in Figure 2-9); 3) The spec.-team searches for relevant information in order to decide who is going to be responsible for a specific bug (shown in Figure 2-10); and 4) A manager or others searching for relevant information on the state of affairs (shown in Figure 2-11). The scenarios presented in Figure 2-8 to 2-11 are based on activities observed in the field-study. They are all highly contextual activities dependent of the role of the actor interacting with the Bug-MOI. From a design point of view, the actors need for search and browsing facilities might be to provide a generic search and browsing function without distinguishing between types of actors and types of situations. Because we in this chapter have chosen to specify a limited set of requirements, this solution seems obvious.

Figure 2-12 illustrates how the requirements chosen in this section could be provided in general. We have, however, maintained the division between the situations in order to stress that the situations are in fact different. In further work on specifying requirements, the fact that the situations and the need for actors to obtain aggregated information are different could lead to radically different browsing and search functions. Searching for matching records is only the basic starting point we have taken here. A natural next step would be to provide different types of graphs, links between mechanisms etc. Here, the spec.-team member assessing state-of-affairs would probably need other functions than a software tester who needs to get an overview of the software errors found.

Actions from the actor(s)	The responds from the Bug-MOI
1) An actor demands “see specific bug”.	The Bug-MOI replies by presenting a search template with entry fields for bug registration id., tester id., bug classification, responsible module, responsible designer, etc.
2) The actor fills in the fields to be searched.	The central bug database is searched for all registered bugs fulfilling the search profile. The search template used and the number of retrieved instances are presented to the actor.
3) If the actor is not satisfied with the found number of bug registrations he fills in new information in the fields and demands “search”.	Se the previous Bug-MOI reaction.
4) The actor asks to “see next bug”.	The complete set of information on the next bug in the list of retrieved records are presented to the actor.

Figure 2-8: A table of the user actions and the matching reactions from the Bug-MOI in a process of searching for information on specific bugs registered.

Actions from the actor(s)	The responds from the Bug-MOI
1) A tester asks to “see similar bugs”.	The Bug-MOI replies by presenting a search template with entry fields for the responsible module and keywords for the bug description.
2) The tester fills in as precise information on the responsible module and the keywords as possible and asks the mechanism to “search”.	The central bug database is searched for all registered bugs fulfilling the search profile. The number of retrieved records are presented to the tester.
3) If the tester is not satisfied with the found number of candidates he or she starts over again by asking to “see similar bugs”.	See the Bug-MOI reaction in step 1.
4) The tester demands “see next bug”	The complete set of information on the next bug in the list of retrieved records are presented to the tester.

Figure 2-9: A table of the typical user actions and Bug-MOI reactions in a process of a software tester searching for bugs with similar characteristics.

Actions from the actor(s)	The responds from the Bug-MOI
1) A member of the spec.-team demands “see roles”.	The Bug-MOI replies by presenting a template with entry fields for roles and actors.
2) The member of the spec.-team fills in the fields of actor or role and demands “see roles”.	The Bug-MOI presents all roles related to the defined actor (or vice versa).
1) A member of the spec.-team demands “see responsibilities”	The Bug-MOI replies by presenting a template with entry fields for modules, tasks, and designers.
2) The member of the spec.-team fills in the fields of designer and/or module and/or task, and demands “see responsibilities”.	All responsibility relations between: • specified designers - tasks, • specified designers - modules, • specified modules - designers, • specified tasks - designers, are presented.
1) A member of the spec.-team demands “see workload”	The Bug-MOI replies by presenting a template with entry fields for designers, tasks, and a deadline.
2) The member of the spec.-team fills in the fields of designer and/or tasks and deadline, and demands “see workload”.	The Bug-MOI presents the man hours to be spend from now until the deadline: • on the specified tasks. • by the specified designers.

Figure 2-10: A table of three typical user actions - Bug-MOI reactions pairs in a process in which the spec.-team attempts to establish the required information for deciding on responsibility. The first supports the spec.-team member in getting information on, for example, who will be platform master in the next period. The next can be used to get information on, for example, who is responsible for the UI-module or which tasks are James responsible for. The intention of the last one is to provide information on how busy a specific designer is going to be, according to the plans, in a forthcoming work period.

The three queries illustrated in Figure 2-10 are all queries that address information we—in this chapter—consider as being outside the Bug-MOI itself. Thus in order to fulfill these requirements the Bug-MOI have to link to other computer based mechanisms providing the relevant information. This will be discussed in section 5.5.

Actions from the actor(s)	The responds from the Bug-MOI
1) An actor demands “see state of affairs”.	The Bug-MOI replies by presenting a search template with entry fields for bug status, bug classifications, module, and designer.
2) The actor fills in the fields he consider relevant and demands “search”. Default for all fields is that all are included.	If no fields have been filled in the actor is requested to do this. If at least of the fields have been filled in, the central bug database is searched for all registered bugs. The number of registrations in the database fulfilling the bug status, the bug classification, the module, and the designer specified is presented.
3) If the actor is not satisfied with the found aggregation number he starts all over by demanding “see state of affairs”.	See step 1.
4) If the actor wants to see details on the bugs aggregated he demands a “see next bug”.	The complete set of information on the next bug fulfilling the requirements on the bug status, the bug classification, the module, and the designer specified is presented to the actor.

Figure 2-11: A table of the typical user actions and the matching reactions from the Bug-MOI in a process of searching for state of affairs information.

The previous four tables illustrated prototypical situations where an actor (a Bug-MOI user) is searching for information in order to establish a basis for taking decisions on how to organize the work. It is important to notice, that the interaction between the actors and the Bug-MOI illustrated are meant as being prototypical, not covering all relevant situations and exceptions from these situations. Further, the search situations illustrated have all been based on the same general structure. This structure is illustrated in Figure 2-12. In a case where the Bug-MOI is going to be implemented, it will be obvious to base the implementation on such a general search structure.

Actions from the actor(s)	The responds from the Bug-MOI
1) An actor evokes the search function.	The Bug-MOI replies by presenting a search template with entry fields for bug registration id., tester id., bug classification, responsible module, responsible designer, etc.
2) The actor sets up search profile by entering fields in the search template.	The central bug database is searched for all registered bugs fulfilling the search profile. The search template used and the number of retrieved instances are presented to the actor.
3) The actor asks to “see next record”.	The complete set of information on the next record in the list of retrieved records are presented to the actor.

Figure 2-12: A table of the generic user actions and the matching reactions from the Bug-MOI in a process of searching for information.

2.5.3. Data structures required in the computational MOI

We have now set up a set of requirements and illustrated how the required functionality could be provided. Thus it is time for a more detailed discussion of which data structures the mechanism must provide access to in order to be able to fulfill the requirements. The access can be provided either by having the structures as an internal part of the Bug-MOI, or by requiring the Bug-MOI to be linked to other computer based mechanisms containing the required structure.

The aim of this section is not to come up with record definitions or other program-like structures, but rather to illustrate which structures are required and what are the central information they should contain.

The structures along which the articulation of software testing activities are performed were already briefly introduced in section 3. Using these and the established requirements for the Bug-MOI, results in a set of structures which must be provided: The registered bugs, the bug classifications, the actors and their roles, the tasks, the work plan and platform periods, and the software architecture.

Several of these structures are, of course, related to each other in different ways. The relations between the structures contained in, or accessible to, the Bug-MOI could be organized as illustrated in Figure 2-12.

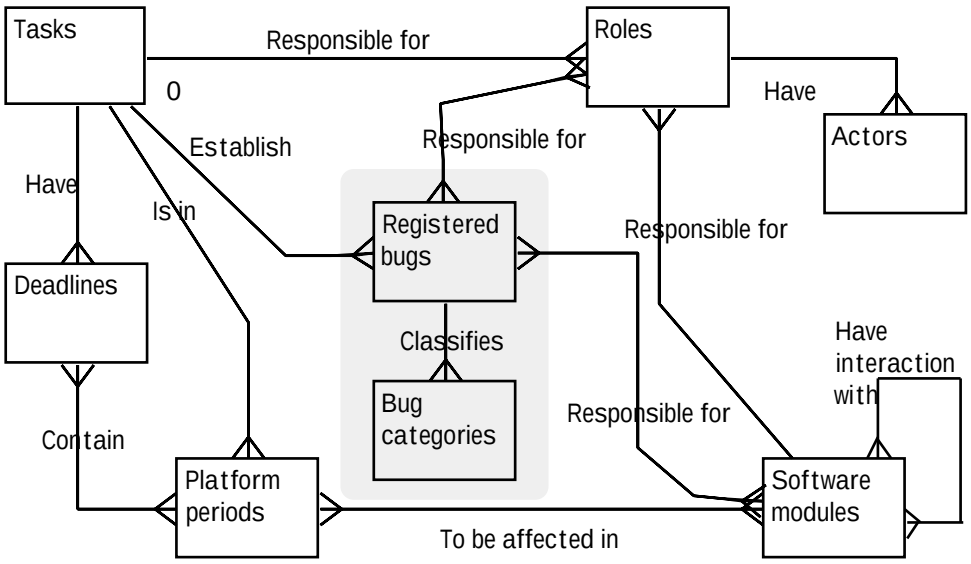


Figure 2-13: A simple version of a entity-relationship diagram illustrating the central structures to be contained or accessed by the Bug-MOI. The boxes are structures and the lines illustrate the relations between the structures. The grayed box illustrates which structures would obviously be inside the Bug-MOI. The forks in the relations indicate a many-relation. For example a bug can have exactly one classification, whereas a bug classification can be used for many registered bugs.

Other work settings could require additional data-structures containing, for example, references to technical resources such as production equipment etc. The following discusses each of the data-structures shown in Figure 2-13:

- The registered bugs.

  The central database in the Bug-MOI should be a register containing a compilation of all relevant information on all bugs ever registered. That is, all information regarding the description of the bug, its diagnosis, and its corrections must be filed here. Further, the classifications and the status of the bug and its correction, and the outgoing relations from the bug structure in Figure 2-13 must be filed. To support the requirements of being able to search for bugs with similar characteristics, the bug description field and the diagnosis description field must partly be based on selections from predefined sets of situations descriptions.

- Bug classifications.

  A generally accepted and commonly understood set of categories for classifying bugs is essential in order to support the distributed registration and classification of bugs, and to delimit the required negotiations on the importance of different bugs. The set of categories must be substantially improved compared to the categories on the existing form (cf. section 4). The classification set will be discussed further in section 5.6.

- The actors and roles.

  The Bug-MOI must have access to descriptions and information on all possible roles to be included in the software testing and correction work. The roles must be defined including obligations, time limitations, etc., and the relations to other structures (cf. Figure 2-13) must be filed. Also information on all the actually involved actors must be accessible. This is, for example, basic information, their involvement in different projects and activities, their main interests and competence, etc.

- The tasks.

  Access to a database covering all tasks, their estimate, their deadlines, a description, a reference to further specification of goal and acceptance criteria, priority, current status, etc. is demanded in order to articulate the testing work. References to responsible roles/actors, the work plans, and an originator of the task must be filed too.

- The work plan.

  A work plan illustrating how all the tasks and roles/actors are related to deadlines (in this case to platform periods) is, of course, a very important tool in relation to articulating the activities. In order to mesh the correction tasks with already defined tasks and to decide on deadlines for the corrections, the Bug-MOI must provide access to such a structure. The work plan structure described here is identical to a combination of the structures tasks, platform periods, and deadlines in Figure 2-13.

- The software architecture.

  The structure of the software modules, their relation to each other, their importance from a product point of view, etc. are essential when deciding on diagnosis of bugs and who to make responsible for the corrections. The

structure must provide information on module name, module priorities, and module status. Further there must be references to detailed specifications, other modules related to a given module, and to who is responsible for designing the module.

Again, this should not be seen as an exhaustive list of all the information, attributes, etc. the required structures should contain. Rather it must be seen as a further refinement of the requirements for the Bug-MOI.

2.5.4. Operations on the data structures

Viewing a computer system as a set of data structures and a set of operations on the data structures, we are now ready to discuss the operations. We have previously identified the following functions of articulating software testing: 1) coordinating activities and tasks, 2) meshing activities, task, and deadlines, 3) relating diagnosis and correction tasks to actors and work periods, 4) allocating resources, 5) monitoring progress and state of affairs, and 6) negotiating classifications, allocations, deadlines, etc. Some of these have been excluded in order to delimit the focus in this chapter (cf. Section 5.1). We have, however, used them as a criteria of relevance when going through the concrete requirements for the Actor—Bug-MOI interaction (cf. Section 5.2), leading to the following identification of which operations must be provided:

Firstly, there must be access to *create new* instances of the structures. For the Bug-MOI this is mainly creating new bug registrations. Creation of new instances of structures outside the Bug-MOI (cf. Figure 2-13) should be available in other computer based mechanisms. Also creation of new bug categories must be available, but some limitation on who have access to this will probably be required.

The next basic operation is a *relate to*. The bug structure must be able to access all relations shown in Figure 2-13. The relations are specifications of: bug classification, responsible software module(s), and responsible role/actor(s).

In order to validate information entered and to setup search requests of different kinds, the mechanism must include a *compare* operation. Furthermore, the Bug-MOI must be able to *update* the bug structure. This will update the actual instance of the bug registration in the bug registration database. Use of the update operation can then—as a next step—trigger other actions, e.g., a *send notification information* (e.g., pass information or return a rejection to a specific actor/role), or a *notify external mechanism* about an update to be made.

In order to provide access to the registered information, the Bug-MOI must include a *read information*. This requires access to *setup query parameters* and *search the database*. Reading information will, of course, only make sense if the information can be made visible to the actor. Thus a *present information* is required. By read information we think of reading a specific (set of) instance(s) of a given structure. The read can be organized either as a read from the registered bug database or the bug classification structure, or it can—if the information is placed

in an external mechanism—result in a *read request* to the external mechanism. This we have discussed further in Schmidt et al. (1994).

If we provide access to the above mentioned basic manipulations on the data structures a first basis for establishing of a first version of the Bug-MOI exists.

2.5.5. Links to external mechanisms

The Bug-MOI must have access to other computer based mechanisms in order to provide the facilities we require. These other mechanisms are not established, and have not been discussed in this chapter. We have, however, chosen to briefly illustrate what kind of linking the Bug-MOI could have to other computer based mechanisms. Analysis of the social mechanisms of interaction found in the field studies showed that even these are linked to each other (see Schmidt et al., 1994).

We have illustrated which computer based mechanisms the Bug-MOI could have access to (be linked to) in Figure 2-14 by re-using the diagram in Figure 2-13. The illustration contains only one of several possible decompositions of the mechanisms.

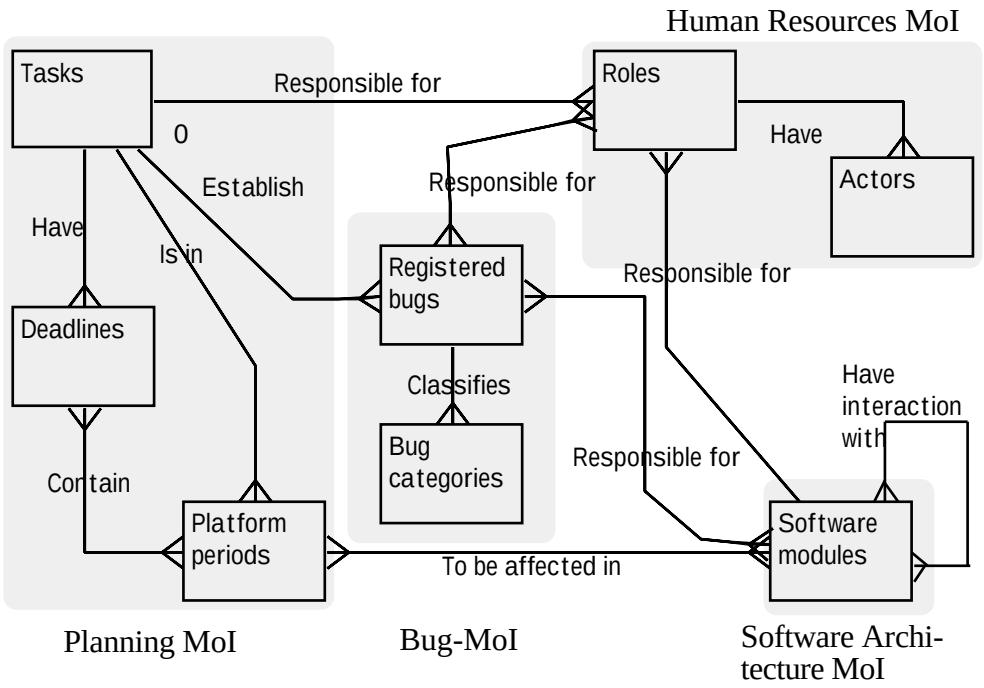


Figure 2-14: An entity-relationship illustration of four computer based mechanisms that, altogether, could support the articulation of software testing. Each of the gray boxes represent a mechanism. The mechanism described in this chapter is the Bug-MOI in the center. The forks in the relations indicate a many-relation.

The Bug-MOI has access to three other computer based mechanisms supporting the work plan aspects, the human resource aspects, and the software modulation aspects of the articulation of software testing.

There are two different types of access to the Planning MOI: One is a *read from* link, i.e., the Bug-MOI can read (or request a read of) the content of structures of tasks, deadlines, and platform period information. The other is a *update* link (used for defining new tasks in the plans). In this case the Bug-MOI can be implemented in one of two ways: It can either write directly in the structures of the Planning MOI, or it can send an update request to the mechanism. The access to information on the roles/actors and their work load are more simple. This is accessed by reading (or request a read of) the content of the structure handled by the Human Resources MOI. The same goes for the Software Architecture MOI. Please notice, that the Bug-MOI can access information on which roles are responsible for which tasks in the current plans both via the link to the Planning MOI and via the link to the Human Resource MOI. The same goes for accessing information on who is responsible for which modules. Here the information can be accessed via the Human Resource MOI or via the Software Architecture MOI

2.5.6. Redesign of the bug form artifact

The age of information processing started out making the mistake of replicating the existing manual administrative systems (Hammer and Sirbu, 1980). Designing computational mechanisms of interaction based on existing paper-based or even computer-based social mechanisms of interaction, does not necessarily mean replicating the existing artifacts and procedures. The artifact used at Foss Electric as a means of supporting the articulation of distributed software testing should therefore not be given the power of computing. If taken seriously, both artifacts and procedures should be made subject to redesign. The purpose of this chapter is to demonstrate the concepts and ideas in going from a qualitative field study to specifying requirements, not to completely redesign the artifacts and procedures used for planning and conducting software testing at Foss Electric. We will, therefore, only sketch one possible solution. We do not take the design process to the stage of designing the user interface. We have chosen to represent the computer-based artifact as a semi-structured message system containing a set of fields to be filled in (Herskind and Nielsen, 1994; Malone et al., 1987), with the additional constraints that semantics is attached to sending messages because of the protocol specified. When designing the user-interface to both artifact and protocol, the form might result in several screens reflecting different roles interacting with the system in different situations.

If we take a critical look at the original paper-form used in software testing at Foss Electric, there are several obvious changes which can be made when turning it into a computer-based form. Some of these changes could, with advantage, be made even to the paper-based form, and others are made due to the fact that a set of linked computational mechanisms of interaction is assumed to be designed. This, for example, results in several fields having default values.

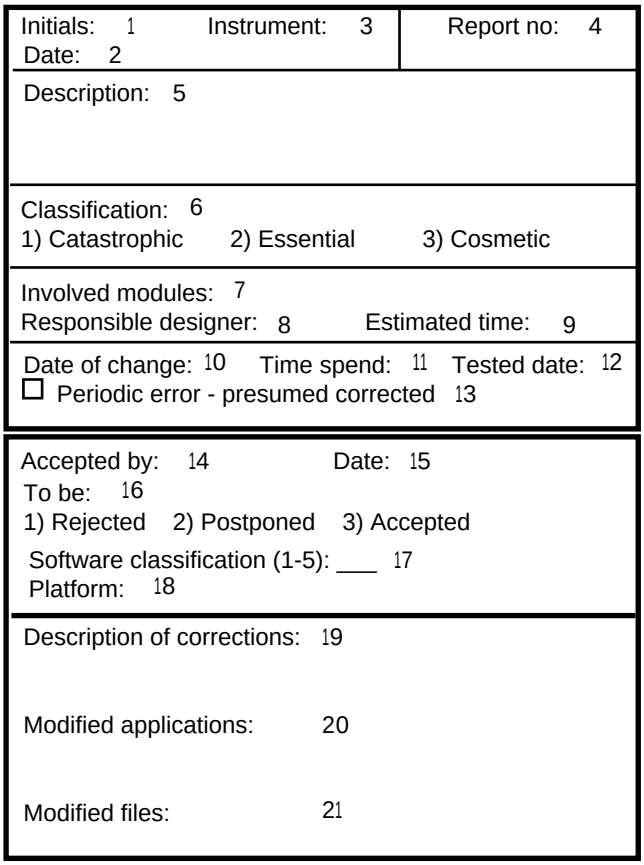


Figure 2-15: The original form with each field enumerated.

Figure 2-15 shows the original form, and Figure 2-16 reviews each field in the original form and discuss the changes decided. The numbers in the left column in Figure 2-16 refer to the numbering of fields in Figure 2-15.

Figure 2-17 shows an initial redesign of the artifact, representing the design decisions discussed in Figure 2-16. In the following we briefly discuss the changes made. The annotated numbers, e.g., *17*, refer to the numbers in Figure 2-15, 2-16, and 2-17.

Redesign of the Bug Report Form	
1	Assuming that the Bug-MOI is linked to the Human Resource MOI, the actor will at some point have logged into the system, and his or her initials will therefore automatically be inserted as the default value. If not appropriate, the initials can be altered
2	The date will be inserted, when the instance of the form is made
3	The tester enters the name of the instrument being tested. If, at a later stage, a repository of pending projects were to be established, this field would naturally be linked to such a repository
4	A unique report number will automatically be generated. In the manual system, only reports of accepted software errors are numbered. This way the last number reflects the total of accepted bugs. In the Bug-MOI simple queries will quickly be able to determine this number, even if all reports are numbered
5	The description of the software error (the phenomenon) detected is a mixture of classification and free text annotations. The Bug Categories data-structure provides a classification scheme of types of software errors. Free text is used as a “other” category. Since both testers and spec.-team members add information to this field, it must be split in two
6	The classification of the seriousness of the software error needs serious redesign. The chosen solution is a two dimensional classification (perspective x importance). Using arguments as in the previous field, the classification of seriousness must be split in two
7	Characterization of the modules involved in testing the error are to be made by accessing the Software Modules data-structure
8	Since the Bug-MOI will be linked to the “Planning MOI”, the responsible designer(s) can be assigned as a default by accessing the relationship between roles and software modules. The default value can be overruled by choosing one or more other designer from a list. Because neither this or field 9 covers the allocation of more than one involved designer, an additional field is needed
9	The estimated time to correct the error is filled in by the spec.-team members
10	The date of change of the software is automatically given the default value of the date the designer reports the changes made
11	The amount of time spent is filled in by the designer
12	The date of the changes being tested is given the default value of the date the designer reports changes made. This implies, together with the default of field 10, that in all other cases than the changes being made and tested the same day, the defaults must be altered
13	If the error is assumed to be periodic, and has been assumed to be fixed, the designer checks in this box
14	The spec.-team member accepting the error form chooses his or her initials from a list. Initials on more than one person can be added
15	The field is default assigned the date the spec.-team receives the form describing the software error
16	The classification of how to proceed further could be linked to the classification of bug categories. Since a fourth category “Postponed indefinitely” is already used at Foss Electric, although it is not on the form, we have chosen to include it
17	This field is redundant, because it covers the same information as field 18
18	This field is actually linked to the Planning MOI. It indicates in which platform the error should be fixed. Current platform period is default value
19	The corrections made can be classified using the Bug Categories structure. Alternatively, free text can be annotated as an “others” category
20	Classification of which software modules and files have been changed inserted by linking to the Software Architecture MOI
21	This field is merged with field 20

Figure 2-16: Redesign of the fields in the original bug form. Numbers in the left column refer to the numbering of fields in Figure 5-9

The major changes made in the re-designed Bug Form (Figure 2-17), compared to the existing one (Figure 2-15) are listed in the following:

- Some fields in the form will upon instantiation or triggered by routing, be assigned default values. This can be accomplished because the Bug-MOI will be one of a set of linked computational mechanisms of interaction. Fields 1, *2*, *4*, *8*, *10*, *12*, *15*, and *18* will be assigned default values. The only one of these fields which are given a default value which can not be changed is field *4*, the report number. In the manual system, the actors needed to maintain an unbroken sequence of report numbers for *accepted* software errors in order to produce statistics. Given the computational power of the Bug-MOI system, this is no longer necessary.
- An additional category, “Postponed Indefinitely”, has been included in field *16*. This category is used for software errors which are recognized as errors, but which the spec.-team chooses to postpone indefinitely. The difference between an error postponed indefinitely and one being rejected is precisely that the former is recognized as a software error, the latter is not.
- Since both software testers and spec.-team members update fields *5* and *6* on the original form, we have chosen to split the two fields in four. This decision is discussed below.
- Field *17* and *18* are, on the original form, used for the same purpose and hence merged into one.
- Given the linking to the Software Architecture MOI, Fields number *20* and *21* are represented in the same field on the re-designed form.
- Field *8b* has been added in order to reflect the practice of using the manual system, where several designers, besides the responsible, can be assigned the task of correcting the bug.
- In order to support the actors in routing the bug reports, fields *22* to *29* have been added. The semantics of the fields are discussed below.
- The history of the form is represented in field *30*. This provides the actor with the possibility of obtaining an overview of who previously have updated the form and when they have done so.

The description of the software error found (field *5*) and the classification of the importance of correcting the software error (field *6*) both need to be redesigned. Firstly, since both software testers and spec.-team members update fields number *5* and *6*, we have chosen to split these fields into four separate fields in the redesigned artifact. The fields *5a* and *6a* are updated by the tester, and the fields *5b* and *6b* by spec.-team members. Secondly, the contents of the two fields needs to be redesigned. The description field (number *5*) is used for characterizing observed phenomena. We propose that a classification structure (contained in the Bug Categories shown in Figure 2-13) is used, characterizing software errors phenomena, e.g., program stopped, window in wrong place, unstable output on tests, etc. In our view, field *6* in the original artifact merges two different criteria: the perspective and the rating of importance. As the field studies

showed, testers categorizing an error as “catastrophic” were almost always overruled (cf. Carstensen, 1994). One way of dealing with this problem is to separate, on the one hand, *why* the tester deems a software error important, and on the other hand, *how* important it is to correct the error. The first could be taken care of by a classification of perspectives or concerns, e.g., maintainability, marketing, stability, safety, usability etc. Classifications of this kind can be build, based on research within software engineering by, for example, standard software quality taxonomies (Boehm, 1981; Fairley, 1985), or the software risk taxonomy proposed by Monarch et al. (1994). As for the importance of correcting the software error, we suggest a scale from 1 to 10, with 10 denoting high importance. Both these classification structures are part of the Bug Categories structure in the Bug-MOI. Using a software risk taxonomy to characterize the perspective and a software error classification for categorizing the observed phenomena should be combined with free text annotations. In classification structures it is important to make sure that there is an “others” or “miscellaneous” category on each level in order to urge the actor to provide as precise a classification as possible (Star, 1994). These “other” fields could contain text annotations, allowing further elaborations.

The original paper-based form does not present the fields in the sequence in which they are filled in by actors in the work-flow. In principle several criteria can be used to formulate the requirements for the design of the Bug-MOI artifact, i.e., the type of data fields presented and the sequence in which the fields are presented to the various actors. One criteria could be to present all fields to all actors in the sequence in which they are filled in. Another criteria could be to present only the most relevant fields as a default based on assessment of which fields are most relevant for each of the roles involved or for each stage in the process. The fields could be prioritized according to the assessed importance and presented in that order. Additionally, criteria regarding the clustering of data elements could be applied. Several data elements could be clustered, for example, all information on actors and roles, information on conceptualizations of the field of work, references to the contents of the field of work, information on decisions made, etc. All these criteria can, of course, be mutually contradictory. The most general solution would be to provide a limited possibility for each actor to create his or her own default views to the data elements. Discussions regarding which data to present and in which sequence are important when designing artifacts to be used. We have, however, deemed further discussion along these lines of only marginal importance in this context, which is one of demonstrating concepts and ideas. We have, in this chapter, simply chosen to change this so the approximate sequence in which the fields are updated is reflected in the sequencing of the clusters of fields filled in by testers, spec.-team members, designers, and the platform master.

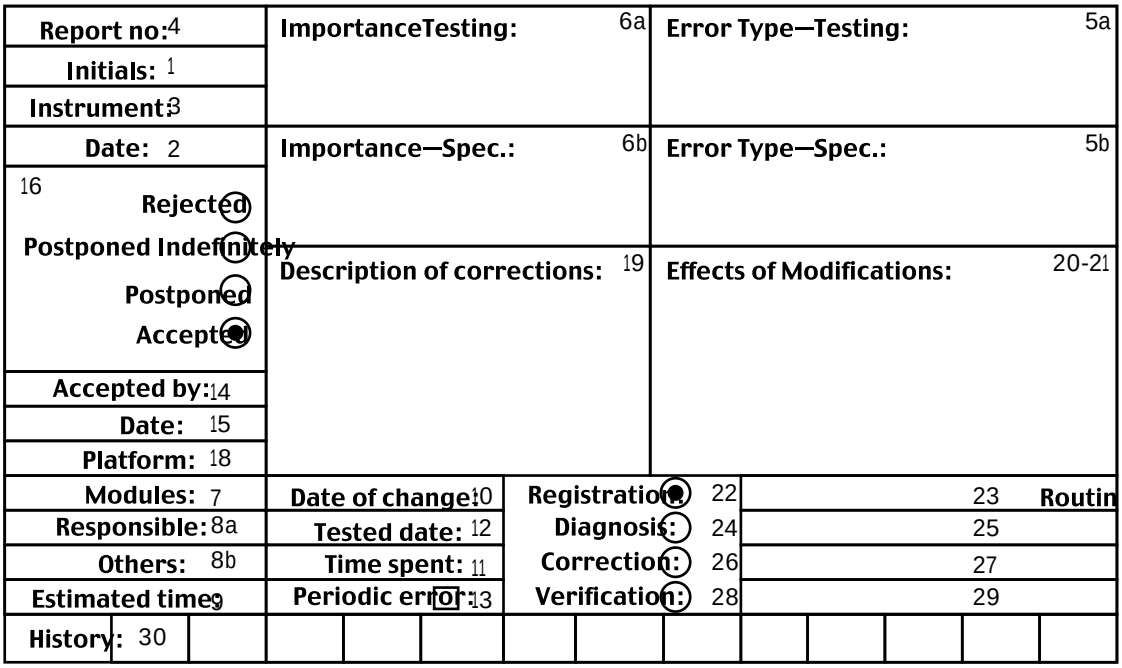


Figure 2-17: Redesign of the Bug-MOI artifact. The software tester uses the default values or fills in fields *1*, *2*, *3*, *4*, *5a*, and *6a*. The spec.-team applies default values or fills in fields *5b*, *6b*, *7*, *8a*, *8b*, *9*, *14*, *15*, *16* and *18*. The software designer(s) correcting the software error applies default values or fills in fields *10*, *11*, *12*, *13*, *19* and *20–21*. All actors use fields *22–30*

Fields number *22*, *24*, *26*, and *28* are included in order to provide the actors with an overview of which stage in the process the Bug-MOI is at. These fields are also used when routing the form from one actor to one or more others, and hence potentially changing the state of the Bug-MOI from, for example, registration to diagnosis. If a person has filled in the appropriate fields needed in order to go from one stage to another, the actor selects the desired phase (fields *22*, *24*, *26*, and *28*). The corresponding routing field (*23*, *25*, *27*, *29*) then shows the possible receivers as a list of roles with the stipulated next step as the default value.

At the bottom of the form, the history of the process is represented by a sequence of fields each containing information on role and date. This way the actor can get an overview of which states the Bug-MOI has passed through before it arrived on his or her desk.

2.5.7. The protocol for the computational MOI

For each form a protocol is stipulated, governing the routing of the form through the four main stages, from a software bug is registered by a tester, to one or more members from the spec.-team diagnosing the bug, over one or more software designers fixing the bug, and lastly ending with the platform master verifying that the problem is corrected. In the following we outline the protocol by discussing general requirements as to how a form can be routed.

The process supported by the protocol consists of the following four distinct phases (see Figure 2-18): Registration, diagnosis, correction, and verification. The software testers are responsible for registration. Software tester is a role which can be played by a number of actors, e.g., software designers, people from the service department, from the quality control department, the quality assurance department, customers etc. Diagnose is taken care of by spec.-team members, a role played by software designers. Correction of software errors are performed by software designers. Verification of the corrections are done by the Platform Master, played by a software designer, and who can delegate tasks to "Deputy Software Masters".

The standard procedure would, in most cases, be that a role sends the form as a request to the next role in the process. If the receiver chooses to reject the request, for example because of incomplete information, the form is returned to the sender (see Figure 2-18).

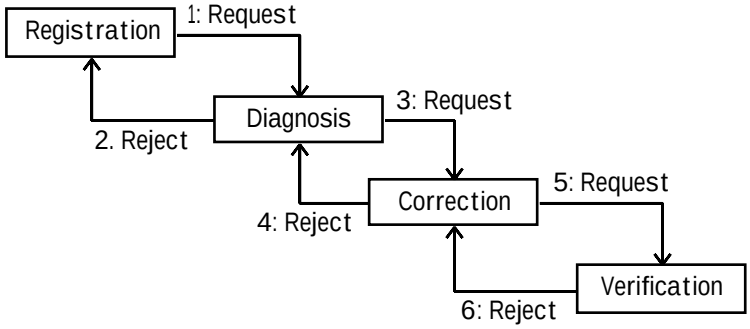


Figure 2-18: The standard protocol. In each of the four stages, the actors sends the form as a request to the subsequent stage, which in turn might reject it.

A form can be sent without any restrictions to roles within each phase, i.e., a spec.-team member forwarding to other spec.-team members, testers forwarding to other testers (see Figure 2-19).

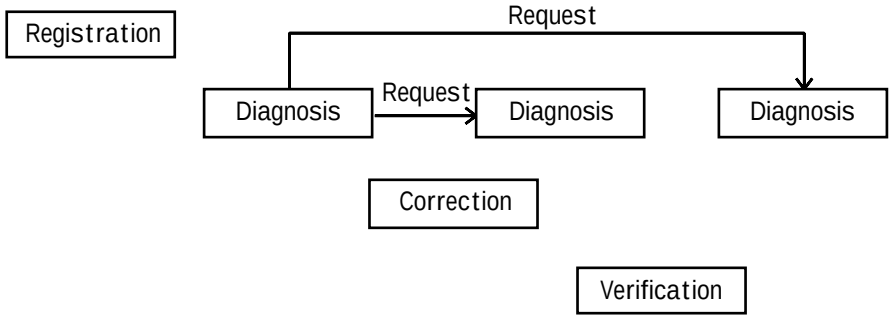


Figure 2-19: Within any stage the actors may send the form as a request to others, who in turn can reject the request.

A sender can choose to skip one stage in the process "downstream". Whenever a receiver is surpassed, he or she is automatically notified. A form can not be sent

"upstream" unless it has been send as a request, which, if rejected, will be returned to the sender (see Figure 2-10). A surpassed receiver, i.e., a receiver who has been skipped "downstream" can choose to claim the form. If, for example, a tester sends a form directly to a software designer, a spec.-team member is notified, and he or she can subsequently choose to "intercept" the request (see Figure 2-20).

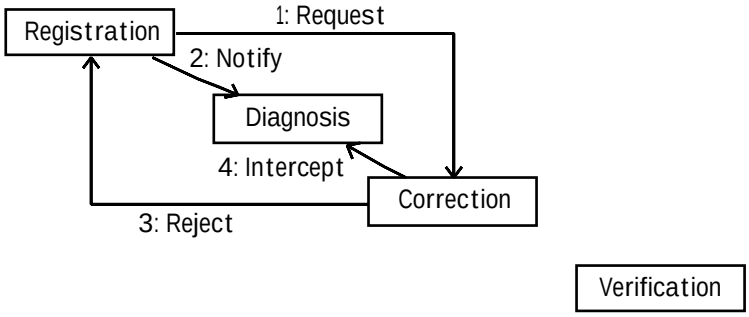


Figure 2-20: The strict sequencing of the four stages can be only be broken by the sender skipping a stage downstream (1). The actors in the stage which has been skipped are notified (2). If the receiver rejects, the form is passed back to the sender. Actors in the stage being skipped can choose to intercept (4).

2.6. Discussion

The aim of this chapter was to illustrate the specification of detailed requirements for a computational mechanism of interaction. These requirements provide input for the refinement and exemplification of the architecture of, and notation for, computational mechanisms of interaction. The results of this can be seen in the following chapter of this book.

The process of establishing the requirements reported in this chapter is the first in which we have used the concept of mechanisms of interaction as a tool for: identifying relevant perspectives to the analyzed work setting, structuring the descriptions of the findings, and guiding the specification of requirements. One of the central questions to address in future work on requirements for mechanisms of interaction, concerns the validity of the approach. We need to investigate to which extent the experiences gained in this specification process can be transferred to other situations. In order to initiate this inquiry, we conclude this chapter with some reflections on the validity of the results in this chapter, and on the process of analyzing a work setting and establishing requirements for computer support.

2.6.1. What has been accomplished?

This chapter has provided a set of general requirements for computer support of the articulation activities involved in software testing, and a detailed description of an existing social mechanism of interaction. Based on this, we have outlined a

computer based mechanism supporting the processes of: 1) ensuring that all registered bugs are treated, i.e., all registered bugs are either diagnosed and corrected, or they are explicitly rejected, 2) providing an overview of the state of affairs of testing and correction activities at any given point in time, and 3) distributed registration, centralized diagnosis, distributed correction, and central compilation of verification information. The computer based mechanism does not provide support for negotiation and communication among the actors, although this is a very relevant requirement.

We have, by presenting a series of interactions between actors and the mechanism, illustrated how a mechanism could fulfill the requirements. The general requirements led to at set of specific requirements concerning the data structures contained in, or accessible to, the mechanism, as well as operations on these structures.

In order to outline a computer based mechanism, subscribing to the general requirements, a number of decisions regarding the detailed requirements and the design, has been made.

Firstly, the structures used for classifying and describing a bug has been redesigned. The classification of a bug is now based on two dimensions: perspective and importance. In classifying the perspective, the actor provides an answer to the question: why does the bug represent a problem? The answers are chosen from a predefined list providing a nomenclature of perspectives, e.g., marketing, maintenance, etc. The importance of the bug is, for each perspective, scored on a scale from 1 to 10. The fields for describing bugs have also been improved, in that they can be based on predefined phrases contained in a classification structure. The refined classification improves the support of the distributed registration, of the diagnosis, and facilitates the negotiation of how bugs are classified and diagnosed. The results of providing improved descriptions of software bugs are increased possibilities for the actors to obtain an overview of the state of affairs and to browse for specific information.

Secondly, the design takes into account that several designers is engaged in a specific correction task. This also improves the support for getting an overview of the state of affairs and for browsing for specific information.

Thirdly, a major improvement concerns routing of the information. The stipulation of the work flow is made active, i.e., when an actor indicates that he or she has concluded a task, the relevant information is automatically passed on to the next actor as a request. The mechanism, however, provides actors with the option of rejecting requests, and actors have certain degrees of freedom regarding who to pass information to. According to Schmidt et al. (1993) is it very important that a computational mechanism of interaction provides a high degree of local control, i.e., the actor must have the possibility to overrule the mechanism. This could, in principle, be used as an argument for allowing actors to route the information to whoever he or she wishes. This would, however, reduce the possibilities for actors obtaining an overview of the state of affairs. One of the crucial features of the Bug-MOI is exactly that it stipulates a protocol,

reducing the complexity of distributed software testing. This requires that data structures always are kept up to date and containing consistent information. We have, thus, decided only to provide a limited set of options for routing.

Another essential requirement for computational mechanisms is to provide actors with the possibility of assessing the status of the mechanism (Schmidt et al., 1993). The mechanism must offer visibility. Because the Bug-MOI protocol stipulates who can send requests to whom at each phase, the actors are provided with various information concerning the state of the mechanism. The Bug-MOI provides the actor with information on which software testing phase it is currently in. When an actor is about to change the state of the mechanism, he or she is presented with the options. Whenever a phase is skipped, the actors responsible for this phase is notified. If no restrictions had been imposed on the process of sending and rejecting requests, this feature would not be so important. Historical information about the states which the mechanism has been in, and when, prior to being received by an actor, is also provided. This feature also facilitates visibility.

2.6.2. Generality of the requirements

Although the primary purpose of this exercise has been to provide an exemplar as input to the following chapter, it has also served the purpose of establishing a more general approach to, and concepts for, understanding the process of designing mechanisms supporting the articulation of complex work. It is, therefore, relevant to reflect upon the validity of the requirements in relation to other work settings and situations, even though this problem is far too complex to be discussed thoroughly in this subsection. The social mechanisms mentioned in the following are all described in COMIC deliverable 3-2 (Schmidt, 1994b).

The need for access to conceptual structures reflecting structures in the cooperative work arrangement and in the actual field of work seems to be a general requirement. The Augmented Bill of Material (ABOM) (Sørensen, 1994a), the CEDAC Board (Sørensen, 1994b), the Construction Note (Andersen, 1994), and Fault Report Form (Pycock and Sharrock, 1994) are all examples of other social mechanisms of interaction containing pointers to structures in the work arrangement and in the field of work. Also, some of the basic operations on these structures (e.g. relating structures to each other, routing information, making others aware of changes in the structures, etc.) are similar in the social mechanisms analyzed.

The support for defining, applying, and refining different types of classification schemes seems to be a general requirement as well. In both the Product Classification Scheme, the Fault Report Form, and the Construction Note classification is essential for the articulation of work. Furthermore, when objects are classified, there seems to be a need for a channel (or structure) through which the classification structure can be negotiated. This was one of the requirements we have chosen to exclude in this chapter. It is, however, a central aspect of articula-

tion work, and thus a central requirement for computational mechanisms of interaction.

The need for stipulation of the work flow seems, not surprisingly, to be a requirement in most of the examples too. Apart from the bug form discussed in this chapter, the Construction Note, the ABOM, and the Fault Report Form also include "build in flow protocols" specified by conventions and organizational procedures. In all these cases, there are also need for actors to be able to deviate from the predefined routing.

Finally, there is the topic of linking mechanisms. In much articulation work, activities such as meshing, relating and coordinating structures are essential. It is, thus, not surprising that mechanisms supporting articulation work must have access (links) to structures within other mechanisms. The linking of different mechanisms is discussed at length in Schmidt et al. (1994). The need for linking could, in theory, be eliminated by supporting articulation work by one big mechanism of interaction. This would, however, conflict with our approach to organizations as multi-facetted and open-ended, and work arrangements as constantly changing open-ended structures (Schmidt, 1994a).

2.6.3. Requirements for the analysis and requirement specification process

The process of analysis, requirement specification, and design, has, to a large extent, been driven by the concepts of social and computational mechanisms of interactions. In the following we reflect upon the process and the usefulness of the concept.

Although articulation work and cooperative work are heavily intertwined, the analytical distinction between the two proved useful. It supported us in explicitly addressing articulation aspects of the cooperative work observed. The concept of social mechanism of interaction supported the identification of possible candidates for computational mechanisms.

One of the problems here were the lack of conceptual support for identifying candidates for computational mechanisms that did not already exist as social mechanisms materialized as a symbolic artifact being used according to organizational procedures and conventions. A functional approach to conventions, procedures, artifacts, etc. used to support the articulation of the work is needed. This could facilitate analyses of which functions of articulation work are needed in specific situations. If an interpretation of a cooperative work setting could result in a set of functions of articulation work, then these functions could be subject for further analyses, amongst others, discussing which part of what functions could be supported by conventions, which by organizational procedures and policies, which by computational mechanisms etc. The elicitation of requirements for computational mechanisms of interaction based on qualitative field studies requires application of additional methods to observing phenomena. Methods, conceptualizations, and techniques supporting the process of analyzing data are needed.

Studying projects which span a considerable amount of time, and which changes from day to day, can not rely on observation alone. Furthermore, as we experienced when observing project work at Foss Electric, for long periods nothing seems to happen. Of course things are happening all the time. The problem is, however, twofold. Either things might not be happening the place we are looking, or things are happening inside the head of project participants. Project participants spent a significant amount of time sitting at their computer doing work alone.

When specifying requirements we experienced the well-known problem of establishing criteria for what is relevant to include. Further research should, amongst others, focus on establishing conceptual support for specifying the allocation of functionality between actor and system (cf. section 5.1). The most important input for the requirements specification process has been our knowledge of structures and processes relevant for articulating the work and an overview of technological options (cf. Kensing and Munk-Madsen, 1993). The dimensions of objects of articulation work and the general requirements for computational mechanisms of interaction presented in Schmidt et al. (1993) has also been a major source of inspiration. The relationships between the conceptual elements in the general requirements must, however, be more thoroughly understood. A concrete example in this paper is the trade off between, on the one hand the need for local control in the routing and, on the other hand the need for consistency in the central database, as discussed in section 6.1.

Outlining the design has first of all been characterized by the difficulties in separating the specification of requirements and designing structures and protocols. As in any design process, theories will only facilitate the process. The fundamental design decisions will inevitably be a matter of making design decisions. Another problem has been how to design a computer based mechanism of interaction taking into consideration how cooperative work will be supported. Supporting the articulation of distributed cooperative work by means of computational mechanisms of interaction should, of course, be conducted in consonance with development of computer support for work itself. Real-life design situations must, of course, also consider how the first design sketches should be evaluated, presented to the actors, etc. This has, however, been out of scope of this work.

Acknowledgments

This research could not have been conducted without the invaluable help of numerous people at Foss Electric, concerning the bug form and its use. Especially Jørn Ørskov and Ole Pflug was helpful. Hans Andersen, Kjeld Schmidt and Carla Simone have been very helpful by raising a number of relevant questions to, and pinpointing weaknesses in, the analysis and interpretation of the findings. The authors would also like to thank Tuomo Tuikka for his participation in analyzing our field study findings and in the specification of the overall requirements for com-

puter support of the articulation of cooperative software testing. All errors in this chapter naturally remain the responsibility of the authors.

The research documented is partially funded by the Esprit BRA 6225 COMIC project and the Danish Technical Research Council.

References

Andersen, Hans: The Construction Note. In K. Schmidt (Ed.): *Social Mechanisms of Interaction*, Esprit BRA 6225 COMIC, Lancaster, England, 1994.

Andersen, N. E., F. Kensing, J. Lundin, L. Mathiassen, A. Munk-Madsen, M. Rasbech, and P. Sørgaard: *Professional Systems Development — Experience, Ideas, and Action*. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

Boehm, B. W.: *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

Borstrøm, Henrik, Peter Carstensen, and Carsten Sørensen: *Two is Fine, Four is a Mess — Reducing Complexity in Manufacturing*, Risø R-768, Risø National Laboratory, 1994.

Cambell, Robert L.: Will the Real Scenario Please Stand Up, *SIGCHI Bulletin*, Vol. 24(2), 1992, pp. 6–8.

Carstensen, Peter: The Bug Report Form. In K. Schmidt (Ed.): *Social Mechanisms of Interaction*, Esprit BRA 6225 COMIC, Lancaster, England, 1994.

Carstensen, Peter, and Carsten Sørensen: The Foss Electric Cases. In K. Schmidt (Ed.): *Social Mechanisms of Interaction*, Esprit BRA 6225 COMIC, Lancaster, England, 1994.

Carstensen, Peter, Tuomo Tuikka, and Carsten Sørensen: Are We Done Now? Towards Requirements for Computer Supported Cooperative Software Testing. In P. Kerola, A. Juustila, and J. Järvinen (Eds.): *Proceedings of Quality by Diversity in Information Systems Research. 17th Information systems Research seminar In Scandinavian (IRIS 17), August 6–9*, Syöte Conference Centre, Finland, University of Oulu, 1994, pp. 424–438.

Ellis, C. A., S. J. Gibbs, and G. L. Rein: Groupware: Some issues and experiences, *Communications of the ACM*, Vol. 34(1), 1991, pp. 38-58.

Fairley, R.: *Software Engineering Concepts*. McGraw-Hill, Singapore, 1985.

Flores, Fernando, Michael Graves, Brad Hartfield, and Terry Winograd: Computer Systems and the Design of Organizational Interaction, *TOIS*, Vol. 6(2), 1988, pp. 153-172.

Gelperin, D, and B Hetzel: The Growth of Software Testing, *Communications of the ACM*, Vol. 31(6), 1988, pp. 687-695.

Hammer, Michael, and Marvin Sirbu: What is Office Automation? In *Proceedings of Proceedings. First Office Automation Conference*, Atlanta, Georgia, March, 1980.

Harrington, Joseph: *Understanding the Manufacturing Process. Key to Successful CAD/CAM Implementation*. Marcel Dekker, New York, 1984.

Heath, Christian, Marina Jirotka, Paul Luff, and Jon Hindmarsh: Unpacking Collaboration: the Interactional Organisation of Trading in a City Dealing Room. In G. De Michelis, C. Simone, and K. Schmidt (Eds.): *ECSCW '93. Proceedings of the Third European Conference on Computer-Supported Cooperative Work, 13-17 September 1993, Milan, Italy*, pp. 155-170, Kluwer Academic Publishers, Dordrecht, 1993.

Helander, Martin, and Mitsou Nagamachi (Eds.): *Design for Manufacturability — A Systems Approach to Concurrent Engineering and Ergonomics*, Taylor & Francis, London, 1992.

Herskind, Steffen R., and Henrik S. Nielsen: *Designing Mechanisms of Interaction for Emergency Management*, Centre for Cognitive Informatics, 1994.

Ishii, Hiroshi, Kazuho Arita, and Takashi Yagi: Beyond Videophones: TeamWorkStation-2 for Narrowband ISDN. In G. De Michelis, C. Simone, and K. Schmidt (Eds.): *ECSCW '93. Proceedings of the Third European Conference on Computer-Supported Cooperative Work, 13-17 September 1993, Milan, Italy*, pp. 325-340, Kluwer Academic Publishers, Dordrecht, 1993.

Johnson, Philip M., and Danu Tjahjono: Improving Software Quality through Computer Supported Collaborative Review. In G. De Michelis, C. Simone, and K. Schmidt (Eds.): *ECSCW '93. Proceedings of the Third European Conference on Computer-Supported Cooperative Work, 13-17 September 1993, Milan, Italy*, pp. 61-76, Kluwer Academic Publishers, Dordrecht, 1993.

Karat, Claire-Marie, and John Karat: Some Dialogue on Scenarios, *SIGCHI Bulletin*, Vol. 24(4), 1992, pp. 7.

Kensing, Finn, and Andreas Munk-Madsen: PD: Structure in the Toolbox, *Communications of the ACM*, Vol. 36(4), 1993, pp. 78-85.

Malone, T. W., K. R. Grant, K. -Y. Lai, R. Rao, and D. Rosenblitt: Semistructured messages are surprisingly useful for computer-supported coordination, *TOIS*, Vol. 5(2), 1987, pp. 115-131.

Monarch, Ira, Marvin Carr, Suresh Konda, and Carol Ulrich: *An Interactive Computational Approach for Building a Software Risk Taxonomy*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213, USA, 1994.

Myers, Glenford J.: *The Art of Software Testing*. John Wiley and Sons, New York, 1979.

Parnas, David L.: Software Aspects of Strategic Defence Systems, *Communications of the ACM*, Vol. 28(12), 1985, pp. 1326–1335.

Patton, M.Q.: *Qualitative Evaluation Methods*. Sage Publications, USA, 1980.

Pycock, James, and Wes Sharrock: The fault report form. In K. Schmidt (Ed.): *Social Mechanisms of Interaction*, Esprit BRA 6225 COMIC, Lancaster, England, 1994.

Schmidt, Kjeld: Modes and Mechanisms of Interaction in Cooperative Work. In C. Simone and K. Schmidt (Eds.): *Computational Mechanisms of Interaction for CSCW*, pp. 21-104, University of Lancaster, Lancaster, England, 1993.

Schmidt, Kjeld: The Organization of Cooperative Work — Beyond the 'Leviathan' Conception of the Organization of Cooperative Work. In T. Malone (Eds.): *Proceedings of CSCW '94.*, Chapel Hill, North Carolina, ACM Press, New York, N.Y., 1994a.

Schmidt, Kjeld (Ed.): *Social Mechanisms of Interaction*, Lancaster University, Lancaster, 1994b.

Schmidt, Kjeld, Hans H. K. Andersen, Peter Carstensen, and Carsten Sørensen: "Linkable Mechanisms of Interaction as a Representation of Organizational Context," in J. Bowers (ed.): *A Conceptual Framework for Describing Organisations*, COMIC, Esprit Basic Research Project 6225, Computing Department, Lancaster University, Lancaster, U.K., 1994. - [COMIC Deliverable 1.2. Available via anonymous FTP from ftp.comp.lancs.ac.uk].

Schmidt, Kjeld, and Peter Carstensen: *Arbejdsanalyse. Teori og praksis [Work Analysis. Theory and Practice]*, Risø-M-2889, Risø National Laboratory, 1990.

Schmidt, Kjeld, Carla Simone, Peter Carstensen, Betty Hewitt, and Carsten Sørensen: Computational Mechanisms of Interaction: Notations and Facilities. In C. Simone and K. Schmidt (Eds.): *Computational Mechanisms of Interaction for CSCW*, pp. 109-164, University of Lancaster, Lancaster, England, 1993.

Simone, Carla, Kjeld Schmidt, Betty Hewit, and Alberto Pozzoli: *An Architecture for Malleable and Linkable Mechanisms of Interaction*, WPCS-94-6, Centre for Cognitie Science, 1994.

Star, Susan Leigh: Personal communication. Risø National Laboratory, Denmark, 1994

Strauss, Anselm: Work and the Division of Labor, *The Sociological Quarterly*, Vol. 26(1), 1985, pp. 1-19.

Suchman, Lucy A.: *Plans and situated actions. The problem of human-machine communication*. Cambridge University Press, Cambridge, 1987.

Sørensen, Carsten: The Augmented Bill of Materials. In K. Schmidt (Ed.): *Social Mechanisms of Interaction*, Esprit BRA 6225 COMIC, Lancaster, England, 1994a.

Sørensen, Carsten: The CEDAC Board. In K. Schmidt (Ed.): *Social Mechanisms of Interaction*, Esprit BRA 6225 COMIC, Lancaster, England, 1994b.

Sørensen, Carsten: The Product Classification Scheme. In K. Schmidt (Ed.): *Social Mechanisms of Interaction*, Esprit BRA 6225 COMIC, Lancaster, England, 1994c.

Chapter 3

Towards an architecture based on a notation for mechanisms of interaction

Carla Simone, Monica Divitini, and Alberto Pozzoli

University of Milano

3.1. The requirements of a notation for Computational Mechanisms of Interaction

A good design of an artificial object starts with the definition of the requirements the object must satisfy. This general rule holds also for the notation for Computational Mechanisms of Interaction, the topic of this chapter. The basic requirements of the notation have been illustrated in Chapter 1; we will revisit them with the aim of identifying the properties the notation must posses in order to fulfill the requirements.

Let us recall from Chapter 1 the two basic definitions:

A mechanism of interaction (MOI) is a protocol, encompassing a set of explicit conventions and prescribed procedures and supported by symbolic artifact with a standardised format, that stipulates and mediates the articulation of distributed activities and thus reduces the complexity of articulating distributed activities of large cooperative ensembles.

A computational mechanism of interaction (C-MOI) is a computer artifact that incorporates aspects of the protocol of a mechanism of interaction so that changes to the state of the mechanism induced by one actor can be automatically conveyed by the artifact to other actors in an appropriate form as stipulated by the protocol.

The definition one immediately can derive two initial requirements of the notation.

First of all, the notation must allow for a *smooth transition from the description of a MOI* that has been identified in an analysis of a cooperative ensemble *to the specification of the corresponding C-MOI* to be designed. This requirement points to a very well known problem in systems development: the communication of ‘system requirements’ by the analysts to the designers who are supposed to transform them into ‘system specifications’. Unless supported by a suitable notation, the communication between these two classes of experts is one of the main

sources of misconceptions, misunderstandings, and mis-interpretations in systems development and therefore one of the most frequent reasons of systems failure. In fact, generally these people do not share the same skills and perspectives regarding the system and the setting where the system is expected to be put to work.

Secondly, the notation must be able to *express which notifications should take place* in presence of specified conditions or internal states of the various components of the C-MOI. Moreover, the notifications may be required *at any level of granularity* within the C-MOI.

An additional set of requirements for the notation are derived from the requirements that MOIs and C-MOIs (the objects of the development process) must satisfy. In Chapter 1 these requirements have been described in terms of flexibility, both as malleability and as partial specification, and in terms of linkability. Let us consider all of them in turn and order.

Malleability requires that the notation supports the *manipulation of the C-MOI description* both at the time of its definition, i.e., in the form of permanent modifications, and during its operation, i.e., in the form of local control of execution. Moreover, as already emphasised in Deliverable 3.1 (Simone and Schmidt, 1993, Section 2.5.4), *effects of such modifications* should not only be timely *notified* to the involved cooperating people but also *anticipated and controlled* in order to avert possible catastrophic effects on their behaviour.

Partial specification requires that the notation provides facilities that make the C-MOI *support articulation work even if not all the information* about its activation *is available*. As stated in Chapter 1, this might depend on a lack of information on the part of the designer. Here, we want to focus on another motivation. C-MOIs - as almost all computer artifacts - cannot be designed under the assumption that the behaviour of the users is completely mediated by the C-MOI. Indeed, it is likely that some actions or negotiations are performed outside the system and that the latter is requested to guess and reconstruct them from their consequences. For example, an assignment of an Activity to an Actor can happen in a face-to-face conversation; this fact may be recognized when the Actor starts the Activity by possessing all the access rights to the required resources.

Linkability requires that the notation is able to *express notification policies across C-MOI* specifications.

Last but not least, the requirements implied by flexibility and linkability should be met in presence of another basic requirement: *the notation should be visible* to the users. In this context, visibility denotes not only available but especially understandable, useful and usable by the users, whereas users may be end-users, experts on organization and articulation work, as well as people providing technical environments for the development of C-MOIs. The challenge is thus that *the notation must be understandable, useful and usable to all types of users*.

The above requirements for the notation can be put in relation to specific properties of the notation, as described in Fig. 3-1. Here, only the more relevant connections are represented, even if each property can only fulfill the corresponding requirements in presence of all the other properties.

Notational properties vs. notational requirements	**Semantic level**	**Formal semantics**	**Modularity**	**Triggers**	**Recursiveness**	**Openness**	**Primitives**
Analysis vs. design	X	X				X	
Internal notification			X	X			
Manipulability	X	X	X				X
Impacts of changes	X	X					X
Partial specification	X	X				X	
Linkability			X	X	X		X
Usability, usefulness	X		X				X

Fig. 3-1. How the Notation Properties contribute to the fulfillment of the Notation Requirements. The plain lines concern the notation expressive power while the dotted part (on the right) concerns the additional supports (i.e., the primitives) of the notation.

The characterisation of the appropriate semantic level of the notation is based on the identification of the minimal set of Objects of Articulation Work (OAW) and of the relations among them, that have been discussed in Chapter 1. Section 3.3.3 illustrates in detail how the notation represents the OAW and the relations among them. The *semantic level of the articulation work* is one of the basic characteristics of the proposed notation and distinguishes it from the many proposals for the definition of environments where to develop CSCW applications (see Chapter 1): these environments are based - in the most advanced proposals - on languages at the semantic level of the manipulation of objects without proposing a set of atomic concepts that are necessary and (hopefully) sufficient for building the applications, in our context the C-MOIs supporting the articulation work. As shown in the table, the semantic level of the articulation work is fundamental to support the communication among analysts and designers, to allow for the manipulability of the MOI description and to govern the impacts of changes, to support the handling of partial specifications, and finally to make the notation usable to all users. In fact, the semantic level defines a finite and powerful framework where the various users can represent the phenomenology of

the articulation work in a uniform way since it creates a shared understanding across all the activities required by the articulation work.

The notation should posses a *formal and visible operational semantics* for many reasons. First of all, a formal and visible operational semantics is required in order for the notation to serve as a reliable communication medium between the various types of users. Further, a formal and visible operational semantics is required to manage modifications in a sound way since only a formal semantics can support consistency checks and provide for the evaluation of the impacts of the modifications. Moreover, the possibility of handling partial information presupposes objects and relations that are unambiguously specified. And finally, since the MOIs have been defined in terms of protocols, that is, in terms of dynamic behaviour, the notation should be able to represent control flow through suitable and *formally defined control structures*. The formality and the visibility of the underlying operational semantics is again a crucial aspect of the proposed notation as it constitutes one of the pillars on which the support to the articulation work is based.

Since the notation is composed of a selection of basic elements (the OAW, their relations and the formal structures to describe the dynamic aspects of the protocols), its richness and usability depend on the flexibility by which these various components can be combined to describe and to link different protocols. Moreover, the modifications can be done more easily and less expensively if the components of the notation can be isolated and substituted, thus reducing the impacts the changes may have on other components. This can be achieved if the notation shows a *great degree of modularity* at the representation level as well as at the level of the operational semantics.

The presence of constructs to express internal conditions and the related *triggers* to be issued, both to the other objects of the same C-MOI and to the linked C-MOIs, is the only way to fulfill the requirement that C-MOIs should be ‘active’ in relation to their users and their computational environment - active in the sense that “*changes to the state of the mechanism induced by one actor can be automatically conveyed* “.

The recursive nature of the MOIs (i.e., a MOI can be the ‘field of work’ of another MOI) has been mentioned in Chapter 1. In order to represent this feature the notation should allow for the recursive invocation of C-MOIs within other C-MOIs. Then *recursiveness*, together with modularity and triggers, allows the notation to represent in a quite flexible way the linking of C-MOIs, and more specifically, between C-MOIs and their Organisational Context.

By definition, C-MOIs are the computational part of the MOIs. The relationship between a MOI and its computational part can be expressed in terms of an ‘embedding’ that makes computational aspects interleave, so to speak, ‘in a continuum’, with aspects that cannot be represented within a computer. Then the notation should be *open to* represent these *non-computational parts* as specific points in the control flow where the outside context (either the user or other applications) should be invoked so that it takes the control up to the point where

the control can be again resumed by the C-MOI. This feature contributes to the already claimed smooth transition from the description of MOIs into the specification of their computational part during system development. Moreover, it can be seen as an additional way to reach flexibility through a type of partial specification that can be handled and solved mainly by the context.

Finally, the notation should support the definition of *primitives* that make the definition, manipulation and use of the C-MOIs for the articulation work more easily performed by the users. These primitives exploit the above mentioned properties of the notation, and provide additional services to its use.

3.2. The layered structure of the notation

The malleability requirement needs a further specification in order to define the notation in an appropriate way. More specifically, it is necessary to specify which type of malleability the notation for C-MOIs should provide.

Let us explain this point through an example taken outside the domain of articulation work in order to reduce the complexity of the exemplification. Let us suppose that the computer artifacts (the protocols) needed by a user to accomplish his work are a collection of integer arithmetic expressions. These latter are constituted of the four basic operations composing variables and integer constants and of parentheses following the well-known syntax. Let us call this language ARITH-Lang. The operational semantics (OP) of ARITH-Lang is the well-known computation process of the arithmetic expressions. This process is based on a precedence relation among the operations stipulating that division and product have priority over sum and subtraction. During his work, the user might need to adapt these artifacts to different conditions of use or to more demanding requirements; then, expressions need to be a malleable computer artifact. We can distinguish among different degrees of malleability that are represented as different ‘levels’ in Fig. 3-2. In that figure, the text in italic presents some examples of the malleability of the expressions at each level, while the remaining part can be viewed as a general framework (a sort of template) whose contents will be considered also when the computer artifacts under concerns will be the C-MOIs.

Let us comment on Fig. 3-2. At the α-level, the expressions are given and can be used through the instantiation of the variables and the computation of the instantiated expression. The adaptation (LOCAL CONTROL) concerns just the current instance. At the β-level, the expressions are modified (PERMANENT CHANGES) to meet additional requirements: then these modifications will affect any future instance of the new expressions. At this level, one can also modify the operational semantics, i.e., the computing process. At the γ-level, the adaptation is more radical since the definition of a more powerful grammar (generating REAL-Lan) allows for the construction of a more powerful set of expressions (not just the integer, but also the real ones!).

α-LEVEL: protocol instance

USE:	instantiate instance:	*assign integer values to the variables of an expression*
	activate instance:	*start computation*
LOCAL CONTROL:		
	- enforce a state:	*assign an arbitrary value to a sub-expression*
	- change OP:	*modify the priority (e.g., ÷ prec * prec - prec +)*
	- change instance structure:	*add a sub-expression since to solve the current problem the user needs an expression just similar to one of the available expressions.*

β-LEVEL: protocol

USE:	- define protocols:	*define new expressions (using ARITH-Lan)*
	- select OP:	*e.g., standard priority*
PERMANENT CHANGES:		
	- modify protocol:	*change the structure of an expression*
	- change OP:	*modify the priority (e.g., ÷ prec * prec - prec +)*

γ-LEVEL: grammar

USE:	- define / modify grammars:	*define new production rules to enrich ARITH-Lan, e.g., with new types of variables and operations (real numbers with exponential, square root and the like), thus obtaining REAL-Lan.*
	- assign OP:	*e.g., computing process based on a selection of algorithms for computing the not trivial operations*

Fig. 3-2 A schema illustrating the various aspects of malleability on a simple example.

This simple example highlights some aspects that will be captured by the notation for C-MOIs supporting articulation work.

1) Malleability can be considered at three levels, with quite different specialisations. At the α- and β-levels we can distinguish between a routinely use of the protocol (USE) and an exceptional situation where some adaptation is needed (LOCAL CONTROL and PERMANENT CHANGES). This distinction disappears at the γ-level where the notation is used just to generate and modify grammars.

2) It is possible to associate to the same language different operational semantics (here, different priorities). Therefore, a complete definition specifies both of them and the modifications can affect either of them.

3) Allowing for LOCAL CONTROL and PERMANENT CHANGES requires that the operational semantics is defined in a highly compositional way (i.e., at a

the level of granularity of each single component), since the components constituting either the protocols or their instances can be ‘rearranged’ at any time.

4) Each level defines the ‘space of possibility’ of the lower level. At the α-level one can instantiate the protocols provided by the β-level whereas the γ-level provides the grammars usable at the β-level to define such protocols. Up to this point the use of the notation requires of the user merely the capability of combining and selecting predefined items by following the rules associated to the grammars and the related semantics.

5) The space of possibility within which grammars can be defined at the γ-level is defined by the ‘semantic level’ pertinent to the goal of the notation. In the simple example, the semantic level is constituted by integer and real variables/constants and a finite collection of integer and real operations. More generally, the γ-level must contain the information about the ‘row’ components from which the grammars can be constructed. But also in the simple example, there is the need of choosing a predefined set of operations that is sufficient to generate all the other functions required by the application domain (as no computational notation can contain an infinite collection of items!). In the case of expressions the definitive answer comes from computability theory. In the case of C-MOIs the choice is determined by an iterative process exploiting both empirical evidences and studies on formal languages, in order to establish a set of Objects of Articulation Work and of Formal Structures which is sufficient to generate the C-MOIs supporting articulation work.

6) The three levels require different user skills and roles within the organization. The α-level and β-level are typically needed by end-users whose role implies the use and the design of C-MOIs for articulation work, respectively. The γ-level is typically the level of the ‘application designer’. In our framework, they are the people in charge of the definition of the grammars needed by the end-users to build protocols and in charge of the possible enrichment of the basic components for grammars definition. While the construction of grammars could be performed by end-users with a specific modelling attitude and capability, the design of new components is a pure programming activity that requires specific technical skills. This activity would, in principle, no longer be necessary, if and when the set of identified basic components is sufficient for generating all the C-MOIs needed for supporting articulation work, which would leave the notation under the full control of the end-users.

The above considerations lead to conceive of *a notation for C-MOIs layered in three levels*, according to the schema of Fig. 3-2 The layered structure of the notation is represented in Fig. 3-3. In addition, the picture shows that the notation does not live in a vacuum: indeed, as a software device it is embedded in a context constituted by, most importantly, the User Interface and the Field of Work. The nature of the links between the notation and its context will be discussed in more details later on (see section 3.4, 3.5 and 3.7), once the notation is specified. However, at this point some considerations are still possible.

First of all, the notation is fully disjoint from any notation used in the design of User Interface. This means that the proposed notation has not to be assessed from the user point of view. To the contrary, it has to be evaluated from its capability of supporting malleable and linkable C-MOIs. In this view, the dotted connections have to be interpreted as requirements that the notation 'imposes' to the UI design: namely, that the objects populating the UI must show the same degree of malleability and linkability as the notation does. Moreover, that the UI design should take into consideration the different types of users that typically access each level (see point 6) above).

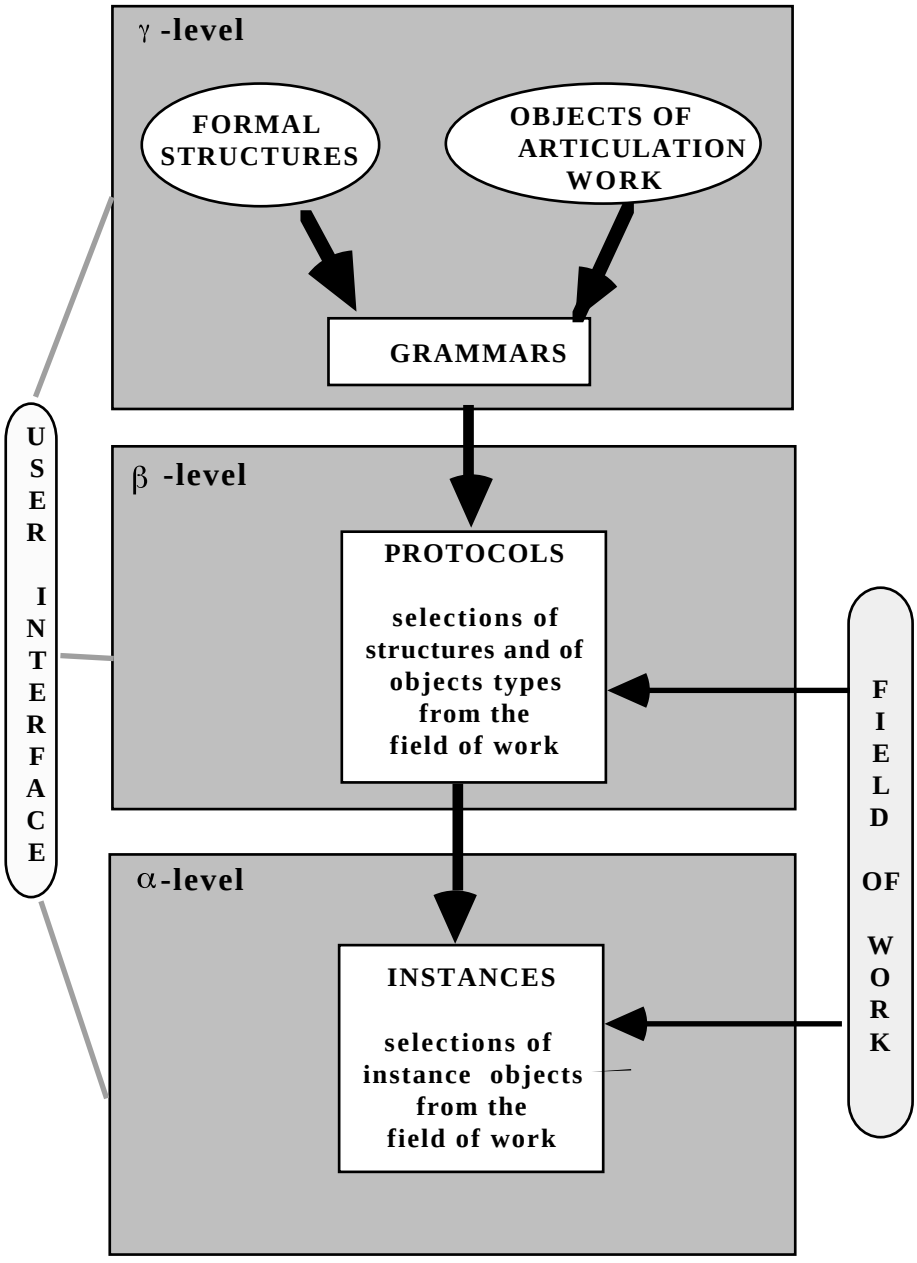


Fig. 3-3 The layered notation and its context.

The right hand side of Fig. 3-3 shows the connections between the notation and the field of work (FoW).

At the γ-level there are no connections with the FoW. In fact, building a grammar means selecting a language to define a class of C-MOIs: more specifically, which components constitute the C-MOI together with their structural interrelation (e.g., in order to model workflows, classification schemes, etc.), and the possible alternative behaviour of the C-MOI through the definition of operational semantics associated (compositionally) to the components of the grammar. The grammars are independent of the FoW in that they simply define the expressive power of the languages potentially used to define of specific protocols.

By contrast, the definition and use of protocols are related to the FoW in many ways. Firstly, the FoW determines the grammar to be used in relation to the complexity of the protocol to be designed. For example, if concurrency is needed then a grammar providing this feature has to be selected, if not it is possible to use a less powerful one. Secondly, the objects of the FoW are related to the objects of the articulation work both as ‘types’ and as ‘instances’ at the β and α levels, respectively. Within protocols, “types” are imported from the FoW together with the related “methods” that are conveyed to the “instances” in the standard way. These aspects will be clarified during the presentation of the notation and again in section 3.7.

The γ-level is motivated also by the purpose of linking C-MOIs. In fact, linking means building a new C-MOI from existing ones, by means of suitable laws of composition. Then, the new C-MOI is generated by a new grammar that is the composition of the grammars of the source C-MOIs (see section 3.6). Therefore, linking can be performed just at the γ-level, i.e., at the level where grammars are under concern.

The next three sections are devoted to the detailed presentation of the notation, starting from the γ-level. Some parts are quite technical, as they are the most abstract level of the (software) specification of the notation to be refined and used in the implementation of the demonstrators. In any case, a textual explanation is provided so that the formal details can be read just by who is interested in the formal specifications.

3.3. γ-level notation

The γ-level concerns the creation and modification of grammars for specifying the infinite variety of protocols necessary at the β-level. In order to formalise the creation and modification of grammars the notation makes use of two general purpose formalisms:

a) **BNF**, as a meta-notation useful to express the productions of the grammars. In this meta-notation, productions take the form

left-hand-side ::= right-hand-side

START is conventionally the initial symbol; / denotes alternative; * denotes iteration; [] denotes optionality; . denotes concatenation/sequence.

BNF has associated a set of meta-functions to manipulate the structure of the productions: ADD(production); DELETE(production); MODIFY(production) with the obvious meaning.

b) **First Order Predicate Logic**, as a language for expressing ***constraints*** (e.g., to attach to any item of the notation a predicate to characterise the space of possibility of its use); the sub-language of **Propositional Calculus** is used to express ***conditions*** (e.g., properties of the state of a protocol).

In addition, the notation contains formalisms specific to the design of C-MOIs:

c) a set of **Basic Elements** which define a closed space of possibility from which each component of the grammar can be chosen.

d) a **Malleability function** (Mall) (see Section 3.3.4)

e) a **Linkability function** (Link) (see Section 3.6)

As described in the previous sections, there are two types of basic elements: the *objects* related to the dimensions of the articulation work (OAW); and the *formal structures* (such as graphs, Petri Nets, etc.) to express the relationships among the above objects. Every basic element has associated its (set of possible) *operational semantics.* Then, a grammar is created or modified by selecting its components from the set of basic elements together with their operational semantics.

According to the definition of mechanisms of interaction as an artifact together with the related procedures and conventions, the set of formal structures contains two type of items: the **relational structures** to describe the procedures and conventions and the **active artifacts** to represent the information contained in the artifact of the MOI. The former are described in the following section, while active artifacts are presented in section 3.3.2.

The operational semantics of the basic elements is described through two formal languages depending on whether the semantics requires to represent concurrency or not. In the latter case we adopt a Pascal-like language; in the former a CSP-like language (Hoare, 1985).

The fragment of Communicating Sequential Processes (CSP) we will use contains the following elements:

PREFIX e → a : an event e is a prefix of (must occur before) an action a;

EVENTS P ? x : input event, that is the current process inputs a value in the variable x from process P;

P ! v : output event, that is the current process outputs the value v to the process P

EXIT : occurs in a parallel composition and causes its immediate termination;

ERROR : is a catastrophic event that stops any computation; it can be a prefix of a recovery action;

CONTROL x := v : the assignment to a variable x of a value v;
(| P_i)* : alternative composition of the processes P_i ;
[|| P_i] : parallel composition of the processes P_i ;
μX. P → X : iteration of a process P expressed as tail recursion;
P ; Q : the process Q starts when the process P is successfully terminated.

An input (output) event within a process happens just when the symmetric output (input) event can happen in the corresponding process: the result is a synchronization that realises a transfer of information. In other words, the simultaneous execution of P ? x by the process Q and of Q ! v by the process P is equivalent to the assignment : x := v in Q.

3.3.1. Relational Structures

The field studies (reported in Chapter 2 and in Deliverable 3.2) show that the possibility of representing relationships among entities is central to a notation for the design of MOIs: e.g., causal relationships among tasks in workflows, part-of relations in classification schemes and so on. The literature provides several types of formalisms that serve exactly the purpose of expressing such relationships in an unambiguous way. Here, the attribute *unambiguous* is fundamental. In fact, we want to refrain from building ‘yet another in-house notation’, with the additional problem of defining its semantics in an unambiguous way. Our choice is then to exploit existing formalisms whose semantics is well-understood and supported by a sound theory, leaving to the designers of the User Interface the identification of more user-friendly strategies for collecting information from and presenting it to the user, if it is the case.

Now the question is: among the many existing formalisms which ones provide a framework that is necessary and sufficient to our purposes? We look at a minimal set for many obvious reasons, last but not least the cost of their implementation.

The identification of this minimal set depends on two factors: on the one hand, the capability to represent the MOIs that the field studies are identifying and to fulfill the requirements of the C-MOIs that are possibly derived from such MOIs (see the example in Chapter 2). On the other hand, the selected structures must possess the properties mentioned in section 3.1 (Fig. 3-1) so as to allow the most powerful definition of the primitives that are necessary to make the C-MOIs malleable and linkable. In other words, this minimal set is the joint result of both empirical and theoretical investigations: a challenging process whose final outcome will finally reduce the gap between these two worlds.

From the considerations above it follows that the set of relational structures presented here below have to be understood as an intermediate proposal, subject to modification as soon as new requirements are identified in the empirical or theoretical studies. Moreover, a full description of these formal structures is out of

the scope of the present work, as it can be retrieved from the literature. We will present just the information necessary to give the flavour of what we want to have implemented and to understand the examples of C-MOIs specification that will illustrate the use of the notation.

In the current stage the proposal contains the following relational structures:

1) Labelled-Graphs, as a very general purpose formalism to represent non-deterministic relationships.
2) Labelled-AND/OR-Graph and Labelled-Basic-Petri-Nets, as means to represent causal relationships in presence of concurrency.
3) Labelled-High-level-Petri-Nets, as a means to represent systems with memory, i.e., systems whose (concurrent) behaviour depends on some previous transformation of the involved data.

All of them are grounded on a sound mathematical theory providing algorithms for animation, simulation and analysis that can be exploited in the implementation of the primitives. All structures have associated labelling functions to express the interpretation of their constitutive elements, mainly in the set of Objects of Articulation Work (OAW) (see section 3.3.3).

The current set of relational structures is the following:

Relational-Structures :: = L-Graph / AND_OR-L-Graph / L-Petri-Nets

These structures are defined on the basis of some service entities defined to support the definition of the other structures and of their semantics.

0) Service Entities

Let in-arcs(v)/out-arcs(v) be the set of arcs entering/leaving a node v.

The entity Nodes defines the type of nodes that will be used in subsequent structures:

Nodes ::= OR-nodes / AND-nodes / Transition-nodes};

v Nodes

OP(v) = if v OR-nodes then ONE-OF(out-arcs(v)) else out-arcs(v)

The following entity describe the characteristics common to all classes of (labelled) Petri-nets that are a class of (labelled) bipartite graphs (van Leeuwen, 1990):

mL-Bip-Graph ::= <S, T, F, L, A, m_0>, where:

S $\subseteq$ OR-nodes

T $\subseteq$ Transition-nodes

F $\subseteq$ (S x T) (T x S)

A = A_S A_T A_F

A_S, A_T and A_F are set of labels

L= <l_S , l_T, l_F > where :

l_S: S → A_S */ the OR-nodes labelling function/*

l_T: T → A_T */ the transition-nodes labelling function/*

l_F: F → A_F */ the arcs labelling function/*

m_0 : S → Tokens

where Tokens ::= {0,1} / INTEGER / {exp | exp some-Algebra}.

'Tokens' specifies the type of marking associated to the nodes in the initial configuration of the current mL-Bip-Graph. There are three type of possible markings where OR-nodes are interpreted either as conditions ({0,1}), or as counters (INTEGER), or finally as memory (the nodes have associated algebraic expressions).

A special type of Petri-net is the Automaton, that will be used to build more complex nets by composition:

Automaton is a mL-Bip-Graph A = <S, T, F, L, A, m_0,>

where t T in-arcs(t) = out-arcs(t) = 1

i.e., no concurrent transitions can be represented

and Tokens ::= {0,1}.

The relational structures are described here below:

1) Non-Deterministic Relationships

This is the classic definition of a graph that can carry labels over both arcs and nodes:

L-Graph ::= <V, E, L, A>, where:

V $\subseteq$ OR-nodes

E $\subseteq$ V x V

L= <l_V, l_E> where :

l_V: V → A_V */ the nodes labelling function/*

l_E: E → A_E */ the arcs labelling function/*

A_V, A_E are set of labels that can be selected in the OAW.

Let g be a generic L-Graph:

OP(g) ::= if g = NIL then STOP else
OP(L(head(g))) ° OP(L(OP(head(g))))
° OP(sub(target(OP(head(g), g))))

where: ° denotes the composition (sequential execution) of functions;
head(g) gives the initial node of the graph g;
target(e) gives the target node of the arc e;
sub(v,g) gives the sub-graph of g whose initial node is v.

This is the traditional way of recursively traversing an OR-graph, possibly executing an action at each node and arc. Here the action is defined as the execution of the OP of the label associated to arcs and nodes. This simple case illustrates how the semantics of a labelled structure is composed in an high modular way of the semantics of the structure and of the semantics of the domain of interpretation (i.e., labelling).

2) Causal Relationships In Presence Of Concurrency

In order to be able to represent concurrency, AND-nodes have to be added to the traditional structure of OR-Graphs (Nilsson, 1980):

AND/OR-L-Graph ::= <V, E, L, A>, where:
$V \subseteq$ OR-nodes AND-nodes
$E \subseteq V \times V$
$L = \langle l_V, l_E \rangle$ where :
$l_V: V \rightarrow A_V$ */ the nodes labelling function/*
$l_E: E \rightarrow A_E$ */ the arcs labelling function/*
A_V, A_E are set of labels that can be selected in the OAW.

Petri-nets are a well recognized formalism to represent concurrency. There are different classes of Petri-nets (Bernardinello and De Cindio, 1992):

L-Petri-Net ::= Basic-nets / High.Level-nets
Basic-nets ::= SA-nets / P/T-nets.

Place/Transition nets (Reisig, 1985) are a class of Petri-nets widely used in real applications:

P/T-nets are mL-Bip-Graph PT = <S, T, F, L, A, m_0 >
where Tokens = INTEGER.

Since modularity is one of the key-point of the proposed notation Superposed Automata nets (De Cindio et al., 1982) belong to the selection of the relational structures:

SA-nets ::= [$name_1$: $Automaton_1$ ||.... || $name_n$: $Automaton_n$]
where $name_i$ are names of objects that can be selected in the OAW;
$Automaton_i$ are automata A = <S, T, F, L, A, m_0,> where A_S and A_F are empty sets:
|| is the parallel composition a' la CSP (i.e., synchronisation of input/output events);

High-Level-Nets (Jensen and Rozenberg, 1991) allow one to represent control flow with memory:

High.Level-nets are mL-Bip-Graph PT = <S, T, F, L, A, m_0 >
where Tokens = {exp | exp some-Algebra}
(not further described here since they are not used in the examples).

We are not showing the semantics of the structures representing concurrency: we just emphasise that the semantics of concurrent systems can be different in relation to the strategy adopted in executing the sets of concurrent actions:

- *full concurrency* semantics: all possible actions in one shot;
- *step* semantics: any subset of possible actions in one shot. This subset can be identified by means of arbitrary criteria (priority, common property of the labels, and so on).
- *interleaving semantics*: just one action at the time selected in a fully non-deterministic way.

Details about this and formal definitions can be found in (Pomello et al., 1992) in relation to Basic Petri-nets languages. These various semantics can be formulated equivalently in all formalisms representing concurrency.

Why are two formalisms needed to represent concurrency: graph based and Petri-nets based? The answer is that each of them is based on a different mathematical theory and provides alternative means for proving system properties. The unification of such formalisms (and of the other ones not considered here) is a crucial open problem in concurrency theory. This, however, is beyond the scope of the COMIC project.

We conclude this section by mentioning two other formalisms to handle concurrency and distributed systems that are presently under investigation as possible additional candidates, namely the **π-Calculus** (Milner et al., 1992) and the **Actor Model** (Agha and Hewitt, 1987) These formalisms provide the capability of combining various components in a more flexible way through a 'run-time' identification of the components to be synchronised with (as opposed to the situation, e.g., in SA-nets, where the combination is defined 'compile-

time'). This indubitable advantage is paid in terms of a more complex notation and, above all, of the difficulty to define a semantic level, i.e., a set of constructs, appropriate to a notation that has not to be conceived of as a programming language. An example of use of the π-Calculus can be found in Appendix 1 of this deliverable.

3.3.2. Active Artifacts

The definition of MOI reported in section 3.1 assigns a relevant role to artifacts since they, due to their symbolic nature and standardised formats, contribute together with procedures and conventions to stipulate and mediate the articulation of distributed activities. Moreover, the definition of C-MOI requires that such artifacts play an *active role* in the articulation work since they must incorporate aspects of the protocol of a mechanism of interaction so that changes to the state of the mechanism induced by one actor can be automatically conveyed by the artifact to other actors in an appropriate form as stipulated by the protocol.

In order to fulfill the above requirements, the notation should provide a structure able to represent structured information and the capability to notify to its environment the appropriate information when some predefined internal states meet specific conditions. This structure, called *active-artifact*, is defined on the basis of a frame like formalism (Nilsson, 1980) as follows:

<artifact-name>
ACCESS: update: X; read: Y;
TRIGGER: <triggering-condition,triggering-mode, trigger>*
<slot1-name> : slot1-type;
.....
<slotN-name> : slotN-type

where triggering-mode = {broadcast, destination}, i.e., the notification may be selective or be received by any object prepared to receive it.

ACCESS:update: X; read: Y and TRIGGER: <triggering-condition,triggering-mode, trigger>* describe the interface of the artifact from/to its environment: X and Y denote the sets of roles having the read and update right, respectively .

The behaviour of an instance of an active-artifact can be described by the following standard 'demons' associated to it: they are described in our CSP-like formalism.

μUPDATE : ONE-OF(X) ? slot-name* → UPDATE

μREAD : ONE-OF(Y) ! < slot-name, slot-value>* → READ

μTRIGGER : triggering-condition(<slot-name, slot-value>*) →
(triggering-mode = broadcast →
broadcast ! <slot-name, slot-value>* →TRIGGER
| triggering-mode = compute(destination) →
destination ! <slot-name, slot-value>* → TRIGGER)

The behaviour of the above demons can be described as follows.

The first two are listening to the environment in order to catch update/read requests: if there is one, then the update/request is performed. Then the process iterates indefinitely.

The third one is checking if one of the predefined triggering conditions applies to the appropriate parameter(s), then the related value(s) is either broadcasted to the environment, or sent to a destination, this destination being computable from the slot-values contained in the artifact. Then the process iterates indefinitely.

In passing, there are a lot of possible variations of these very simple behaviours: for example, more that one condition/request can be present at the same time. The literature provides different policies based on ‘fair’, non-deterministic, or parallel behaviours. Moreover, both triggering-modes and triggers can be more complex than the simple issue of slot-values. According to our notation style, the set of different operational semantics can be attached to the active artifact as a space of possibility available when the grammars are defined.

It may be possible to conceive of a demon symmetric with respect to the above third one, namely a demon able to listen to external triggers that generate some changes in the content of the artifact. This is ‘computationally’ quite possible but would allow the artifact to be not only active but also ‘too intelligent’. In fact, reacting to a trigger would require a knowledge about the outside world that is not reasonable for something that, basically, is a data structure. The above situation can be simulated easily in the way described in Fig. 3-4, where the process on the left interprets the original trigger and generates an update request to the artifact that generates the output-trigger to a new process that, possibly, interprets it in a domain dependent way:

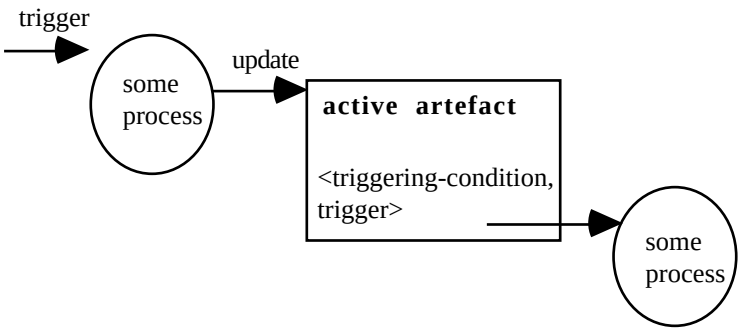


Fig. 3-4. How input triggers can be simulated by the notation.

Other artifacts/MOIs can interact with an active-artifact by means of the following expressions, once they are adequately inserted in their description:

<artifact-name> ! <value>* to modify values
<artifact-name> ? <variable>* to get values
<artifact-name> ? <notification>* to listen to notifications.

As an example of active-artifact, let us consider the Bug Report Form described in Chapter 2 (Fig.2-4). It can be represented as an active-artifact as follows:

<BUG_REPORT=BR>
ACCESS: update, read:
{tester, spec-team, responsible designer, platform master};
TRIGGER: {update(tester), spec-team, send(BR); .....>}
identification: initials : actor;
instrument: tech.resource;
date: time;
report #: integer;
classification: {catastrophic, essential, cosmetic};
correction-info: modules: tech.resource;
responsible des.: role;
estimated time: time;
correct.-to-do: accepted by: role;
date: time;
state: {rejected, postponed, accepted};
s/w classif. : (1-5):
platform #: integer;
correct.-done: description: text;

applications:	info.resource
files:	info.resource.

The BR is active in the sense that it recognises some types of updating, and reacts by sending some information to a specified role. Obviously, actions that require a choice by some role (as explained in Chapter 2) cannot be represented by this type of triggers, unless the trigger is a simple sending of information - as a reminder - to the role who has to take the decision.

Another artifact/MOI can interact with BR by means of expressions like the following ones:

BR ! <value>* , BR ? <variable>*, BR ? <notification>* where variables and values are structured according to the slots constituting the artifact. A more precise definition of this example will be given in section 3.4 (Fig. 3-8).

3.3.3. Objects of Articulation Work (OAW)

In the following we describe the Objects of Articulation Work (OAW). Their description is given in a frame-like notation, i.e., in terms of a list of pairs <attribute-name; attribute type>. The choice of the attributes is based on the characterisation described in Chapter 1 (Fig. 1-1) and on the set of operations on the objects, as shown in Fig. 3-5 (it is just recalled from Chapter 1).

Nominal		**Actual**	
Objects of articulation work	**Operations with respect to objects of articulation work**	**Objects of articulation work**	**Operations with respect to objects of articulation work**
Articulation work with respect to the cooperative work arrangement			
Role	assign to [Committed actor]; responsible for [Task, Resource]	Committed-actor	assume , accept, reject [Role]; initiate [Activity];
Task	point out, express; divide, relate; allocate, volunteer; accept, reject; order, countermand; accomplish, assess; approve, disapprove; realized by [Activity]	Activity	[Committed actor] initiate; [Actor-in-action] undertake, do, accomplish; realize [Task]; [Actor-in-action] makes publicly perceptible, monitors, is aware of, explains, questions;
Human resource	locate, allocate, reserve;	Actor-in-action	initiates [Activity]; does [Activity];
Articulation work with respect to the field of work			
Conceptual structures	categorize: define, relate, exemplify relations between categories pertaining to [Field of Work];	State of field of work	classify aspect of [State of field of work]; monitor, direct attention to, make sense of, act on aspect of [State of field of work];
Informational resource	locate, obtain access to, block access to;	Informational resources-in-use	show, hide content of; publicize, conceal existence of;
Material resource	locate, procure; allocate, reserve to [Task];	Material resources-in-use	deploy, consume; transform;
Technical resource	locate, procure; allocate, reserve to [Task];	Technical resources-in-use	deploy; use;
Infrastructural resource	reserve;	Infrastructural resources-in-use	use;

Figure 3-5. Objects and typical operations of articulation work.

Each attribute type carries the operational behaviour associated with it. The operational semantics of the whole object is built up using these ‘pieces of behaviour’.

The attributes describing each object can be classified as follows:

a) attributes identifying and describing the object: they vary from an object to another;

b) attributes representing the relations with the other objects (corresponding to the cross references between objects contained in Fig. 3-5);

c) attributes representing the relationships with the Organisational Context as a meta-level where OAW are *managed*, *defined*, *assigned* and *adapted*: (Fig. 3-6).

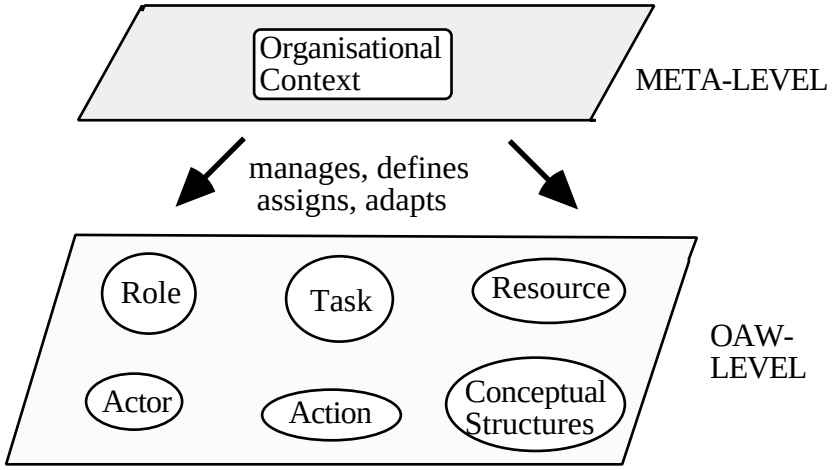


Fig. 3-6. The operations of the Organisational Context that affect the OAW.

The type of these attributes is a pair <Role; policy> with the following meaning: at the meta-level, the Role is responsible of the task evoked in the attribute name and accomplishes it by applying a specific policy. The latter is mentioned at the OAW level since it might depend on the specific object under concern. For example, a generic policy governing the negotiation of the assignment of a Role to a Committed-Actor can be different if the Role is crucial (e.g., the Managing Director) or if the Actor is a specific person (e.g., with a high recognized authoritativeness). From the operational point of view, at the OAW level the policies are, by default, the one associated at the meta-level (i.e., at the Organisational Context level) to the Role mentioned in the pair <Role; policy>; default policies are overridden if more specific ones apply.

d) an attribute, called *awareness*, expressing the capability of each object to convey to its environment information about changes of its internal states (see section 3.1). Awareness is a term taken from Strand 4 and is a way to extend the property of being reactive that we used in introducing active-artifacts (see section 3.3.2). In fact, the type of the attribute named ‘awareness’ is expressed just with a notation similar to the one used for ‘triggers’ within active-artifacts. Specifically, the choice of representing only output triggers is based just on analogous motivations.

The OAW model shown in Fig. 3-5 highlights the symmetric nature of many of the represented cross references. This type of *redundancy* is maintained in the choice *of objects attributes* as a means to allow for the definition of inference rules managing incomplete knowledge. In fact, partial information can come from any OAW and should be adequately recorded so as to allow for the application of the inference rules to generate the information missing in the linked objects. Each single OAW can now be described by means of the notation. Attributes of class c) and d) will be highlighted by shaded names: in the following they are not further commented on since their explanation has being given once for all in the above points c) and d) By contrast, the attributes belonging to class a) and b) will deserve some comment: more specifically, the latter will be confronted with the operations mentioned in Fig. 3-5. The exemplification is based on the case study of Chapter 2.

Objects-of-Articulation-Work ::= **Role** / **Actor** /**Task** /
Activity / **Action** / **Resources**

where

Role =

ATTRIBUTE NAME	ATTRIBUTE TYPE
name	identifier
responsible of	set-of(Resources)
responsible of	set-of(Task)
policies	set-of(rules)
assumed by	set-of(Actors)
defined by	<Role; policy>
assigned by	<Role; policy>
adapted by	<Role; policy>
awareness	<cond;out-trigger>*

The first four attributes define a role through a set of responsibilities with respect to resources and tasks and through a set of policies (rules to be followed). The only attribute of type b) expresses the fact that a role is assumed by actors.

For example, the role ‘Tester’ in the bug form case is defined in terms of responsibility for tasks, i.e., the identification, registration, and classification of bugs, while the role ‘Central File Manager’ is defined in terms of responsibility for a resource, i.e., the Central File.

OP(Role) = “apply rules and policies when matching in the current framework”

This description fully covers the definition of operations mentioned in Fig. 3-5.

Actor =

ATTRIBUTE NAME	ATTRIBUTE TYPE
anagr. data	some strings
assigned to	set of (Role)
committed to	set of (Tasks)
initiator of	set of (Activities)
doing	set of (Activities)
locality	place
skill	some description
assigned by	<Role; policy>
awareness	<cond;out-trigger>*

An actor is a person characterized by its anagraphical data, the set of its commitments, the set of roles it assumes the set of the activities it activates and performs and finally its skills. Being a real entity an actor lives in a place.

The remaining operations mentioned in Fig. 3-5 (e.g., accept, reject) are captured by the policies (protocols) that are associated to the roles the actor assumes.

It may be useful to denote an actor as a member of a set of actors defined either explicitly (actors*) or implicitly by some specific criteria (group). To this aim we introduce the following service object:

Ensemble ::= actor / actors* / group(criteria)
where group(criteria) :: = “all actors satisfying the criteria”
and criteria ::= playing a role/ responsible of a task / actors committed to an activity/ .....”other criteria”.

and

OP(Ensemble) =
if Ensemble = actor then access(actor-info) else
if ensemble = actors* then ONE_OF(actors*) else
ONE-OF(group(criteria)

The elemental operations associated to an actor can be understood as assignments of values to the attributes characterising it: this motivate the use of the function ‘access’ in the above operational semantics.

Task = { t | t ::=

ATTRIBUTE-NAME	ATTRIBUTE-TYPE
name	identifier
what	description
responsible	Role
supervised by	<Role; policy>
trigger(s)	in-triggers
preconditions	test(s)
priority	priority-value
postconditions	test(s)
has allocated	set of (Resources)
realised by	set of(Activity)
criteria of accomplishment	policy
assessed by	<RE ;policy>
approved by	<RE ;policy>
assigned by	<Role; policy>
defined by	<Role; policy>
adapted by	<Role; policy>
awareness	<cond;out-trigger>*

A task is characterised by its content; its triggers, pre- and post-conditions together with its priority. Moreover, as a distributed entity its execution implies a set of activities and its accomplishment has to be defined by a specific policy (e.g., to define how results are collected and considered sufficient to state that the task has been accomplished). Finally, a set of attributes of type 'role' states which role is responsible for this task, which role has to be contacted in case of exceptions (supervisor), and which role is in charge of the declaration that the task has been accomplished in a satisfactory way (assess and approve). The two latter attributes incorporate the idea of closed-loop that characterises the proposal by Medina-Mora and associates (Medina-Mora et al., 1992) and at the same time they provide for a more flexible definition of the involved roles (e.g., the actor who assesses in not necessarily the same as the actor who approves). The policies mentioned govern the flux of information among the various involved roles.

The operational semantics of a task describes how the various policies are applied in the accomplishment of a task.

OP(task) = in-triggers and preconditions and priority =OK →
[[responsible:= OP(ONE-OF(group(X));
[OP (set-of(Activity)) ||
OP(accomplishment-policy); EXIT
||
μT. (| condition → out-trigger → T)*] ;
OP(assessment-policy) ; OP(approval-policy);
set(postconditions); EXIT
||

ERROR → error ! OP (R)]

where X is the following criterion: “playing the Role associated to *responsible*” and R = ONE-OF(group(playing the Role associated to *supervised by*)).

When the in-triggers and the preconditions are satisfied and the priority applies, then a responsible is chosen in the set of people playing the related role, by exploiting the rules and policies pertinent to the committed actor. Then three processes are started in parallel: the first two activate the activities connected to the Task and the related policy for collecting their result, while the third checks the conditions for outputting some output triggers (awareness). Then the assessment and approval policies are activated. All the above processes run in parallel with an error recovery process that is activated in case of an ERROR is raised by one of them. In this case, a call to the supervisor is sent.

Each protocol mentioned within a task can be of any complexity; if it is empty, then it is interpreted as successfully terminated.

Notice that the elemental operations associated to a task are all represented (directly or through policies) but for the ones called ‘divide, relate, express’ that are realised by the primitives ‘define’ and ‘modify’ at the β-level (see section 3.4). At the moment it is difficult to imagine a computational version of ‘point out’.

To accomplish a task, structures of actions are generally required. They are called activity. In the simplest case, an activity contains just an action.

Activity ::= Action / Structure of(Actions)

where Structure-of(Actions) is any of the relational structures (see 3.3.1) labelled in the set of Actions.

There are two types of actions: actions transforming resources and interactions, i.e., exchange of information among interlocutors. The need of these two types of action is substantiated by the case study of Chapter 2.

Action ::= **Perf-Action** / **Interaction**

where

Perf-Action ::= { pa| pa ::=

ATTRIBUTE-NAME	ATTRIBUTE-TYPE
name	identifier
content	description/FoWP
realises	Task
initiated by	Ensemble
done by	Ensemble
in; out; used	sets-of(Resources)
trigger(s)	in-triggers
preconditions	test(s)
priority	priority-value
postconditions	test(s)
state	state-value
temporal constraints	deadline
duration	time-interval
awareness	<cond;out-trigger>*

where FoWP ≠ Field of Work Procedure and
state-value {waiting, ready, started, under-execution, concluded, error} }.

In a performative action its content can be expressed in terms of a call of a procedure of the field of work; as in the case of tasks, by its triggers, pre- and post-conditions together with its priority; and by the involved resources that can play the role of input, output and resources that are simply required for getting the action done. A performative action contributes to the realisation of a task, and is initiated and performed by an actor. Finally, a performative action possesses temporal attributes to express constraints and deadlines, and a state describing its degree of achievement.

Again, the operational semantics describes how the various pieces of information contribute to the behaviour of an action.

OP(pa) = state = ready and in-triggers and preconditions and priority = OK →

```
[  [ OP (content, OP(initiator )) ; EXIT
          ||
            μT. ( | condition → out-trigger → T)* ] ;
  set(postconditions); OP( time-interval); EXIT
 ||
   ERROR → error ! OP (R) ]
```

The behaviour can be described as in the case of tasks, with the following considerations:

OP (content, OP(initiator)) results in a call of a procedure in the FoW or in waiting for an ERROR or for an EXIT stating that the action content has been executed (either a direct input from the user or a signal from the environment where the action is under execution). The value of R is inherited from the tasks the action contributes to realise.

The current semantics exploits (implicitly) the information about the *deadline* as a source of an ERROR if the deadline has expired. A more detailed specification of how to exploit the temporal information is a matter for future work. The same holds for OP(time-interval) that actually is just a 'place-holder' for a function describing the computation involving time to be performed upon a successful accomplishment of the action.

Interaction ::= { a | a ::=

ATTRIBUTE-NAME	ATTRIBUTE-TYPE
realises	Task
sender	Ensemble
content	Info-Res*
receiver	Ensemble
ip	Illocutionary Point
preconditions	test(s)
answer-time	deadline
awareness	<cond;out-trigger>*

The attributes of the interactions are self-explanatory. We want just comment on the fact that when interactions contribute to the realisation of a task, they can be interpreted as an abstraction of a more complex protocol. For example, in the case study of Chapter 2, the sending of information among the various roles can be interpreted - in a simplistic way - as an order to execute the subsequent step. Actually, a deeper analysis has shown that in some cases the exchange of information triggers a negotiation that can end with the request being rejected.

Also the operational semantics of interactions is quite obvious. Sender and receiver can be a single actor or an actor selected out of a group of people.

OP(a) = [preconditions → send(OP(senders), OP(receivers), content); EXIT
|| answer- time has elapsed → OP(answer-time, OP(senders))
|| μT. (| condition → out-trigger → T)*] .

As for the operations mentioned in Fig. 3-5, some are represented as attributes: specifically, make publicly perceptible, monitor, be aware of are modelled through the attribute called 'awareness'. As for 'explain and question', they can be represented through policies and protocols having as object the action itself: in other words, the action is viewed as an 'information resource' of the type described here below.

According to the OAW model, resources are of four different types:

Resource ::= **Info-Res** / **Material-Res** / **Technical-Res** / **Infrastructural-Res**

The set of attributes is common to all types of resources. The distinction allows for the definition of specific values that the attributes can take in each class (as described in the following). Notice the distinction between the attribute 'managed by' and 'governed by': while the second one pertains to the OAW level, the first one belongs to the Organisational Context level. Both of them are needed since they refer to different sets of operations (see Fig. 3-5): the first one refers mainly to the 'actual' column while the second one to the 'nominal' column.

Info-Res ::= { ir | ir ::=

ATTRIBUTE-NAME	ATTRIBUTE-TYPE
name	identifier
type	Info-Res-type
description	text
managed by	Role
allocated to	Task
access rights	<access-type, policy>*
location	space
relation	<Resource-names, rel-type>
state	state-value
governed by	< Role, policy>
awareness	<cond;out-trigger>*

where:

access-type {read, write, copy, move, transfer, load, unload}. This set reflects the set of elemental operations that can be associated to Info-Res.

state-value {a description on the current usage of the Info-Res}.

The specification of an adequate description of the current state of an Info-Res is matter of a future refinement that will take into consideration the requirements stated in (Rodden et al., 1992) and the results of Strand 4 about the management of shared objects.

Info-Res-type {documents, letters, applications, notes, files, memos, reports, drawings}

rel-type {definition, classification ,prototypical, causal, genetic, historical, means/end, part/whole relation; copy of}

In the case of **Material-Res**

access-type {reserve, consume, move, place}

state-value {idle, in use, consumed}

Mat-Res-type is determined by the field of work.

In the case of, Technical-Res and Infrastructural-Res:

access-type {reserve, move, place}
state-value {idle, in use, out-of-order}
Techn-Res-type and Infras-Res-type is determined by the field of work.

The elemental operations associated to a Resource can be understood as assignments of values to the attributes characterising it and by the eventual application of the related policy.

The last object to be considered is called conceptual structure: it is described as categories linked by the appropriate relations.

Conceptual Structure ::=

ATTRIBUTE-NAME	**ATTRIBUTE-TYPE**
name	identifier
categories	objects of AW and FoW
relations	rel-type*
awareness	<cond;out-trigger>*

where rel-type* is as in Info-Res.

An example of Conceptual Structure is the classification of Speech Acts following Searle's taxonomy (Searle, 1975), that defines the values of the attribute called Illocutionary-Point of the OAW called Interaction. Another example is the taxonomy used in systems supporting argumentation (e.g., the rhetorical moves in gIBIS (Conklin, 1988)). These structures are mediating and stipulating capability in the frame of communication pragmatics. Furthermore the field studies reported in the Deliverable 3.2 show the use of several Conceptual Structures to organise objects of the Field of Work. They are mainly classification schemes: the classification of bugs in the case of the Bug Report Form, the classification of types of CAM programs and of the products in the case of the Augmented Bill of Material. The latter are examples of the connections between the Conceptual Structures and the field of work.

This concludes the presentation of the basic elements of the notation. As discussed in Chapter 1, articulation work happens with reference to two additional dimensions: ***time*** and ***space*** but a conclusive interpretation has not been accomplished. For the time being, we envisage the introduction of a *metrics of time* that allows for the handling of typical temporal notions like: duration, milestone, deadline, cycles, urgency, and the like (Egger and Wagner, 1993). In accordance with the spirit of the notation, each component of the temporal metrics should have associated a (set of) operational semantics. For example, in the case of a deadline we can have:

OP(deadline, Ensemble):: = if time has elapsed then **solicit**(Ensemble) /

if time has elapsed then EXIT /
if time has elapsed then **enforce**(behaviour).

Temporal information has to be exploited within both the conditions and triggers used to express awareness in OAW and active-artifacts.

The refinement of the dimension of space will consider the results of Strand 4 about virtual reality.

We end this section by recalling that the current choice of Formal Structures and Objects of Articulation Work is not definitive. In fact, the malleability of C-MOIs will possibly require additions to these basic elements if new mechanisms of interactions cannot be represented by the notation, i.e. this notation should be extensible into unknown situations. The operation for doing this is not at the semantic level of articulation work, and is represented in our setting by the functional ENRICH that makes the set of basic elements open to modifications: ENRICH accesses the programming environment where the new basic elements can be defined and imported in the notation (see section 3.7).

3.3.4. The Malleability function

Besides the basic elements illustrated above, at the γ-level the notation contains a malleability function (Mall) that is used when a new grammar has to be defined or an existing one modified.

Mall takes a grammar as an argument: Mall (grammar) = newgrammar. If the argument is the empty grammar, then Mall denotes a creation; otherwise, it denotes a modification. The function Mall is defined along all components of the grammar. The creation of a new grammar starts by the definition of the production having as left-hand-side the initial symbol START; the modification of a grammar operates on the existing productions by applying the meta-functions mentioned at the beginning of section 3.1. All the grammars already defined are collected into a set called: Grammars-set.

As an example, let us consider the definition of a grammar, called CONV_GR, for the construction of different types of conversation models (the protocols) that are traditionally described by means of a Labelled Graph (Winograd and Flores, 1986). Then the first step of the definition of CONV_GR assigns to the symbol START an L-Graph whose arcs are labelled in the set of the interactions.

START ::= CONV_L-Graph where

CONV_L-Graph = < States, Edges, L, A> where
A $\subseteq$ Interaction and L: Edges $\rightarrow$ A.

The labelling function is not arbitrary. Indeed, it has to follow some semantic constraints: for example, the fact that an *accept, counteroffer*... must follow a *request* /*offer*. This property can be represented by the following L-constraints:

- v {*accept, counteroffer, ...*} , w Path(L-Graph).
wv Path(L-Graph) *request* /*offer* è
(where Path(G) is the set of paths of the graph G; denotes logical implication;
è is the set of Illocutionary Points contained in the interactions constituting w).

As a second example, let us consider the definition of a grammar called WORKFLOW_GR for the construction of workflows that the designer wants to represent by a formalism describing distributed states and actions. Then, the designer chooses to base the description on SA-nets.

WORKFLOW_GR assigns to the symbol START a Labelled SA-net in which transitions are labelled either as a task or an interaction and whose Automata describe the behaviour of the involved roles.

START ::= WF_SA-net where WF_SA-net = [n_1: Aut_1 ||.... || n_n : Aut_n]

and the labelling functions associated to the names n_i and to the constituting Automata Aut_i are as follows:

L_i : Trans-nodes$_i$ → A_i and $A_i \subseteq$ Interactions Tasks
L : {n_i } → Role.

Once a grammar has been defined, then it can be modified through the Mall function, on some of its components, by using alternative basic elements and/or alternative semantics. Below are some examples.

It is possible to change the expressive power of the language generated by the grammar.

Example 1 : Mall(Structures)

This will change the set of structures used in the grammar. E.g., in the CONV_GR one can define:

START ::= CONV_L-Graph / CONV_L-Petri-net
where CONV_L-Petri-net is a Petri-net whose transitions are labelled in A $\subseteq$ Interaction.

This allows the definition of multi-agent distributed interactions in addition to the well-known conversations.

The expressive power can be changed acting also on the labelling functions by modifying the type of the objects constituting the labelling sets.

Example 2 : Mall(L)

For example, in the WORKFLOW_GR one can allows just tasks as labels of the transitions and change the labelling of names from Roles to generic labels. Or one can construct another grammar called WORKFLOW/2_GR by associating to

the SA-net transitions labels of type Actions (both performative actions and interactions).

It is possible to modify just the semantics of a component of the grammar and not the component itself.

Example 3 :
Mall(OP(structure))

This use of the Mall function changes the semantics of the structure. For example if we have a structure for describing concurrent behaviour with associated one of the following semantics (1) execute all possible steps in parallel, (2) execute a subset of the possible sets according to some priority, (3) execute each step in any order, then the current semantics can become one of the other two.

Mall(OP(deadline, Ensemble))

This use of the Mall function alters the semantics of OP(deadline, Ensemble) to change the behaviour of the protocol on a time-out. For example we can change the semantics from a solicit to a new behaviour stating that the elapse of deadline is interpreted as an abort of the current object behaviour or as the fact that one of its states is reached (e.g., as an implicit OK or as an implicit refusal in the case of conversations).

Also the set of constraints (Mall(constraints)) can be changed by means of the following auxiliary functions:

overwrite(constraints, new-constraints) /
add(constraints, new-constraints) /
delete(constraints, new-constraints).

3.4. β - level notation

At this level, the grammars contained in the Grammar-set are ‘visible’ to the users so that they can define the protocols they need. In order to facilitate this the notation at the β-level is mainly constituted by a *set of primitives* supporting the manipulation of protocols. At the β-level the notation contains:

1) a Grammar-set, containing the grammars constructed at the γ-level and the related primitive **access**(Grammar-set).

If we consider the grammars defined in the previous section, then
Grammar-set = {CONV_GR, WORKFLOW_GR, WORKFLOW/2_GR }.

2) a primitive to define the protocols using the grammars of the previous point: namely, **define**-protocol(X) where X is a protocol name.

Protocols are collected in sets whose names are conventionally carrying the type of the protocols followed by the keyword **Protocols**.

For example in the case of conversations we might have:

Conv-Protocols::= {CfA, CfP, CfI, … } where CfA, CfP, CfI are Conv-Protocol names. In this set the Conv-Protocol called CfA can be defined according to the Conversation for Action of the COORDINATOR (Winograd and Flores, 1986), and therefore described by the following transition table:

CurrentState	Speech Act		Next State
()	A	Offers	Offered
	P	Requests	Requested
Requested	A	Counter-Offers	Offered
	A	Refuses	Rejected*
	A	Accepts	Taken
	A	Concludes	Concluded
	A	Delegates	Delegated*
	P	Modifies request	Requested
	P	Withdraws	Deleted*
Offered	P	Counter-Requests	Requested
	P	Refuses	Rejected*
	P	Accepts	Taken
	A	Withdraws	Deleted*
Taken	P	Counter-Requests	Requested
	P	Cancels	Deleted*
	A	Counter-Offers	Offered
	A	Concludes	Concluded
Concluded	P	Refuses	Taken
	P	Evaluates	Satisfied*

* Terminal States; A = Active, P = Passive

Fig. 3-7. the Conversation for Action diagram according to the COORDINATOR.

This is one of the many possible ways of presenting a CONV_L-Graph. One can immediately check that the constraints required by the grammar are satisfied.

In the case of workflows we might have:

WF/2-Protocols::= {BUG-HANDLING, ... } where BUG-HANDLING takes its inspiration from the Bug Report field study reported in Chapter 2 (see Figure 2-7). The protocol can be defined by using WORKFLOW/2_GR . Then BUG-HANDLING is a SA-net whose constituting Automata are listed here below:

```
BUG-HANDLING =
        [Tester-B_H: AUT1 || ST-B_H: AUT2 ||
                        Res.Des-B_H :AUT3 || Pl_Mast : AUT4]
```

The four Automata and their labelling functions are represented graphically in Fig. 3-8. Notice that the transitions representing input/output events have to be considered as simultaneous (as expressed by the CSP-like formalism) and then they should be superimposed in the global net: so, the resulting transitions carry the labels of type interaction.

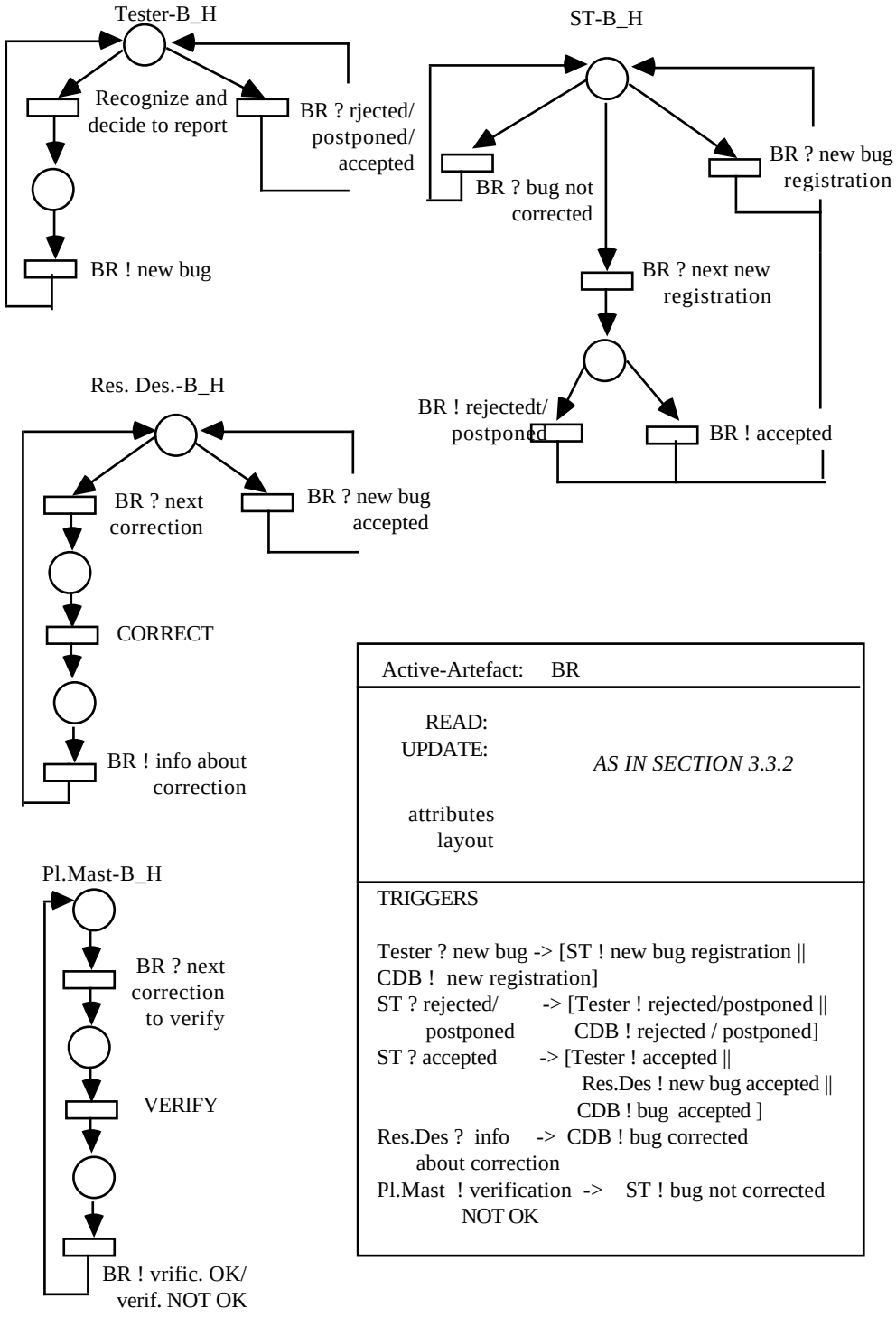


Fig 3-8. The protocol of the Bug_Handling according to the requirements stated in Chapter 2.

The labels shown in Fig. 3-8 are shorthand of the full definition of the labels belonging to the set of actions, associated to each Automaton by the labelling function (and then inherited by the whole net): L: T → B-H_Actions.

B-H_Actions can be defined as in the examples shown in Fig. 3-9 and Fig. 3-10: the first one refers to an interaction while the second refers to a performative action.

ATTRIBUTE-NAME	ATTRIBUTE-TYPE
senders	SPEC-TEAM
content	opinion-request
receivers	Designer
IP	request
preconditions	NIL
answer-time	2 Days
awareness	NIL

Fig. 3-9. An example of Interaction at the β-level.

ATTRIBUTE-NAME	ATTRIBUTE-TYPE
name	Diagnosis
content	Evaluate bug .....
realises	Bug-Correction
initiated by	SPEC-TEAM
done by	SPEC-TEAM
in; out; used	BR; Measures; Bug-classification
trigger(s)	NIL
preconditions	NIL
priority	high
postconditions	NIL
state	ready
temporal constants	in 2 days
duration	4 Hours
awareness	(state=under-exec; output(state, SPEC-TEAM-Chief)

Fig. 3-10. An example of Perf-Action at the β-level.

It is important to notice that the attribute types utilised at the β-level are taken from the Field of Work, as described and discussed in section 3.2. Moreover, the connection with the FoW can be further specified through the enrichment of the *awareness* attribute. In fact, at the β-level it is possible to express conditions on objects of the FoW that trigger notifications to the C-MOI under concern.

Fig. 3-11 illustrates these two different type of connections.

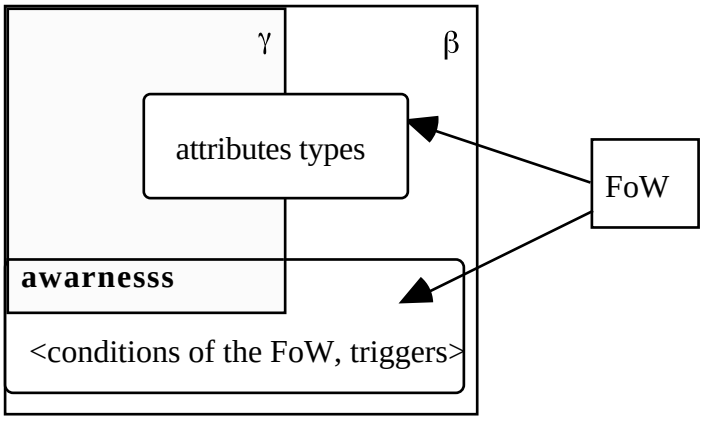


Fig. 3-11. The connections between OAW and FoW at the β-level.

The following primitives allow for the permanent changes of the protocols.

3) a primitive to modify protocols: namely, **modify**-protocol(X, modification-type) where X is a protocol name.

For example, the above CfA protocol can be modified to obtain the CfA protocol implemented in CHAOS (Divitini and Simone, 1994b):

current state	speech-act	next state
()	D Offers	Offered
	R Requests	Requested
Requested	D Accepts	Accepted
	D Counter-Offers	Offered
	D Declines	*End
	R Cancels	*End
Offered	D Declines	*End
	R Counter-Requests	Requested
	R Accepts	Accepted
	R Cancels	*End
Accepted	D Revokes	Revoked
	D Reports	Reported
	D Re-Counter-Offers	Re-Offered
	R Re-Counter-Requests	Re-Requested
	R Cancels	End*
Reported	R Re-Counter-Requests	Re-Requested
	R Report-ok	End*
	R Report-not-ok	End*
Revoked	R Re-Counter-Requests	Re-Requested
	R Cancels	End*
Re-Requested	D Accepts	Accepted
	D Re-Counter-Offers	Re-Offered
	R Cancels	End*
Re-Offered	R Re-Counter-Requests	Re-Requested
	R Cancels	End*
	R Accepts	Accepted

(* Terminal States; R = Referent; D - Doer)

Fig. 3-12. The Conversation for Action according to CHAOS.

Two additional groups of primitives support the design of protocols at the β-level:

4) **animate**(X) and **simulate**(X) provide the user with the possibility of 'playing' with the protocol to perform a sort of test. Simulation is an animation with the additional computation of predefined parameters expressing good behaviours.

5) The remaining two primitives provide the classical support to the management of the various versions of a protocol:

build(history(X))
access(history(X))

We end this section with some considerations about the above primitives and their implementation.

Define and **modify** can be viewed, on one hand, as a means to elicit from the user the information necessary to build or change the current protocol through a nice user interface. This minimalist view merely requires suitable checks of the syntactic definition against the generating grammar (like in compilers). However, the situation is different if these primitives provide some support for the check of correctness of the new protocol against some predefined property. In this case, the notation plays a fundamental role as the formal structures it contains can provide some checking techniques. Just two examples, the wide literature about Petri-nets and the related tools (Brauer et al., 1987) (exploited for example in DOMINO (Kreifelts and Woetzel, 1987)); and the rich amount of algorithms provided by graph-theory to verify graph properties. As an application, consider the proposal contained in (Ellis and Keddara, 1994): here a proof of correctness of a modification of an ICN (Ellis and Nutt, 1980) exploits the theory of graph grammars (Ehrig et al., 1983) as ICN are based on AND/OR-Graphs.

Animate and **simulate** are based on the operational semantics associated to the formalism underlying the protocols. Then the choice of having specified the semantics of each type of basic elements is a fundamental basis towards the implementation of these primitives.

Build(history(X)) and **access**(history(X)) should by implemented by emulating the tools supporting software versioning.

A full specification of the β-level primitives is matter of the research to be undertaken next year.

3.5. α - level notation

At this level, the protocols contained in the Protocol-set are ‘visible’ to the users so that they can define the instances they need. This level is quite standard in many applications: what characterises the proposed notation is the set of primitives that are available to manage the instances of the protocols (local control). Accordingly, the notation at the α-level is constituted by:

1) a Protocol-set, containing the protocols constructed at the β-level and the related primitive **access**(Protocol-set).

If we consider the protocols defined in the previous section, then the set of protocols is defined as:

Protocol-set = {CONV_Protocols, WF_Protocols, WF/2_Protocols}.

Let Y be a protocol name and X be a Y-Instances name:

2) a primitive to define instances using the protocols of the previous point: namely, **define**-instance(X, Y).

Instances are collected in sets whose names are conventionally carrying the type of the instance followed by the keyword **Instances**.

For example, in the case of conversations we have:

CfA-Instances::= {C1, C2, C3, … } where C1, C2, C3 are CfA-Instances names.

At the α-level the relationships with the FoW concern the attributes values. Then the picture describing the relationships at the various levels can be completed as shown in Fig. 3-13.

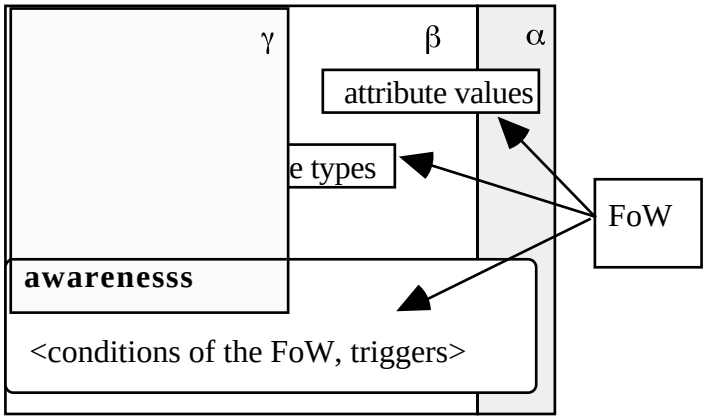


Fig. 3-13. The connections between OAW and FoW at the α-level.

3) a primitive to activate the instance, that is, to put the current C-MOI at work in a specific circumstance: namely, **activate**(X). This primitive is based on the operational semantics of the protocol X is an instance of.

Besides these two primitives, which are standard in many applications, there are two primitives allowing one to modify the behaviour of an instance in two different ways:

4) **modify**-instance(X) allows for structural modifications of the instance: for example, if its source protocol exploits graphs as formal structure, **modify**(X) allows the insertion/deletion of arcs and nodes and/or the change of their labelling functions. These modifications are possible also when the instance has been activated and affect just the current instance while the source protocol remains unchanged. A similar functionality is provided for example in EGRET (Johnson, 1992).

5) **enforce**(X, new-configuration) can be used when the instance has been activated. Its invocation allows for an instance's behaviour to proceed from a new configuration with respect to the current one. For example, if we consider a graph-based formalism, the configuration is made of the current node; in the case of Petri-nets based formalisms, the configurations are the markings; in the case of an active-artifact the configuration is the current set of values. Then, the primitive

allows one to change node, marking and values, respectively, in a way that is independent of the previous configurations.

6) **build**(history(X)) and **access**(history(X)) are primitives that record and make available the various steps the behaviour of the instance went through after its activation. Basically, they handle the history of the execution of X.

7) the last primitive **makePermanent**(X, new-name) allows one to transform an instance into a permanent protocol that will be referenced by the new-name. This operation makes sense after the activation of the primitive allowing for structural modifications that are becoming recurrent so that they can be made permanently available for future uses. A similar functionality is provided for example in EGRET (Johnson, 1992).

A full specification of the α-level primitives is matter of the research of next year.

3.6. The notation and the requirement of 'linking'

The second and most challenging requirement of the notation is the ability to link two or more mechanisms of interaction, that is, building a composed mechanism of interaction from two or more existing ones. Linking extends the modularity of C-MOIs from their internal structure (in order to support malleability) to their environment which is expressed in terms of the other C-MOIs available in the organisational setting. The optimal notation should be able to express both types of modularity (Fig. 3-14).

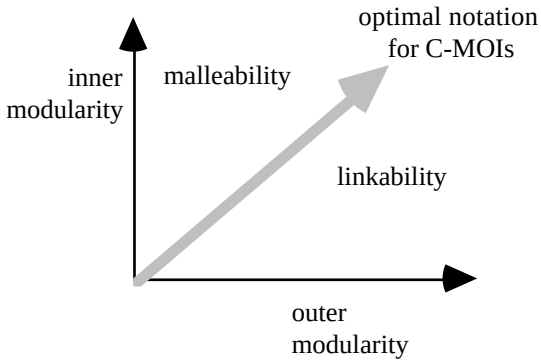


Fig. 3-14. The inner and outer modularity of the notation.

In order to clarify the concept of 'outer modularity', it is useful to take some mechanisms of interaction embodied in the existing applications that have been analysed in Part 3 of D3.1 (Simone and Schmidt, 1993), and 'link' them.

First, let us take a workflow as in DOMINO, where the β−notation generates a type of Petri-net based α-notation of which the elements are: objects, actions and triggering-events, control links with the 'obvious' syntactic and semantic rules. The flow is described in terms of, for example: if an order arrives, then do these

actions in some specified order. Second, let us also take a mechanism based on conversations, as the one described in the previous sections. And third, let us take the mechanism proposed in CHAOS which acts on representations of the organisational context in the sense that it defines roles through policies: in CHAOS, the notation allows for

- the generation of 'role patterns' specifying, e.g., that a role called 'employee in the activity X' has some access rights in relation to some specified objects and some rules to follow in relation to its autonomy (e.g., before modifying an object of type y or accepting a request she has to ask someone for an authorisation),
- and the assignment of patterns to people.

What does it mean to link these mechanisms? For example, we might want to build a framework as in UTUCS_WooRKS (Agostini et al., 1994) with the specific link to the organisational context described above.

Basically, we need a grammar to 'link' notations by means of some 'laws of composition' expressing the various types of linking. For example, the notification of events: e.g., an 'arrival of an order to a person' in the workflow has to be notified to the organisational context' that should handle it as a 'receiving of a request' in a conversation. The resulting semantics could be something like: 'when an order arrives to the person and she has assigned the matching role, then she should ask for an authorisation before accepting to process it according to the workflow'. Moreover, linking the workflow with the communication environment (as in UTUCS_WooRKS) requires a loose coupling at the level of the two notations (basically, they are active in parallel) but has some implications at the level of the primitives. For example, the capability of keeping track of the history of the compound C-MOI requires a filtered access from/to the workflow to/from the communication environment in order to have available the mutually pertinent pieces of information. In this case the composition concerns the behaviour of the primitives as well.

Let us now abstract from the specific examples and consider two generic C-MOIs, M1 and M2, with their related primitives. What does it mean to compose them into a MOI, M3, at the level of their notations as well as of their primitives? Obviously, we presuppose that the user has sound motivations for wanting this composition; we are only concerned with the syntactical aspects together with their operational semantics.

The easiest case is when the composition does not alter the behaviour of M1 and M2. In that case the behaviour of M3 is just the *juxtaposition of the behaviours* of M1 and M2. In other terms, M3 makes **concurrently available** the notations and the primitives of M1 and M2. For example, one could combine a C-MOI based on conversations with one based on a classification schema, that can be seen as completely independent. In this case the true linking is in the head of the user who could be supported with functionalities like 'cut-and-paste' to transfer information from a C-MOI to the linked ones.

A more complex situation is when the composition of the two C-MOIs also involves *a mutual influence on their behaviours*. The mutual influence can be of different types.

Firstly, the composition of M1 and M2 is based on the use of some shared object. It is possible that this object is named and described in different ways within M1 and M2: in this case the composition has to express the **identification of objects** by combining the different views. The handling of this case can profit by the research conducted in Strand 4 about shared objects.

A very interesting case is when the shared object is a policy. This possibility was widely used in the definition of Objects of Articulation Work (section 3.3.3). Policies can be expressed by some rules (constraints) or by some protocol representing another C-MOI. This view of C-MOIs as a special case of objects opens the possibility of introducing *recursiveness* in our notation (Fig. 3-6). This is one point that deserves additional understanding in order to clarify the mutual impacts of recursiveness and awareness (in all its forms) at the modeling as well as at the implementation levels.

Secondly, the composition can be based on the **identification (merging) of actions** (or n-tuples, in a more general but conceptually equivalent case). Identification of actions is traditionally understood as their synchronisation (Fig. 3-15). Actions that are not identified are considered as independent (and then concurrent) in the two source behaviours. Examples of merging of actions have been given in the description of active-artifacts in combination with the procedures they are coordinating (see sections 3.3.2 and 3.4). It is worthwhile to notice that this case is not an example of linking, because the merging merely contributes to building a *single* C-MOI out of the components specified by its definition. However, it is natural to think of the extension of this special case to the case of whole C-MOIs.

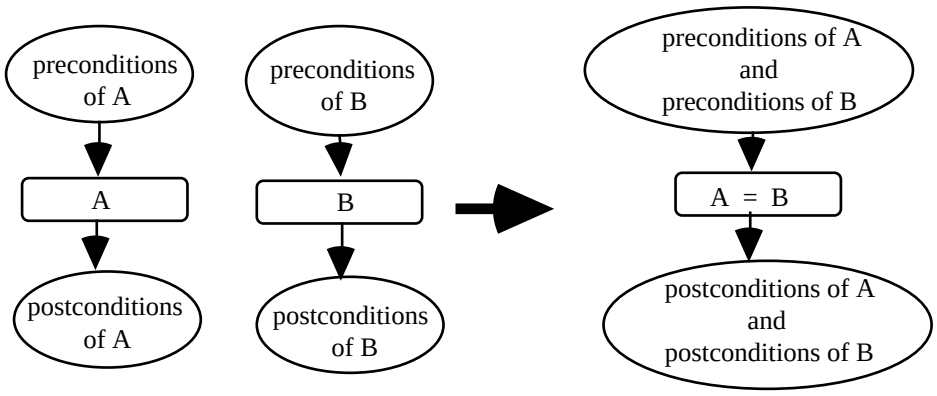


Fig. 3-15 Synchronisation by merging of actions

Thirdly, the composition can be based on the **identification (merging) of states** (or n-tuples, in a more general but conceptually equivalent case). Identification of states is traditionally understood as adding alternatives/non-

determinism (Fig. 3-16). For example, let M1 and M2 be two workflows described in a stated based notation (e.g., graphs or Petri-nets): then, if a state in M1 is identified with a state in M2 the behaviour of the resulting C-MOI contains the identified state that can be followed by either the behaviour of M1 or of M2. The merging of states is a way to express linking when it is based on recursiveness, or more specifically, when in a certain state a protocol has to be invoked and a specific state has to be reached after its execution.

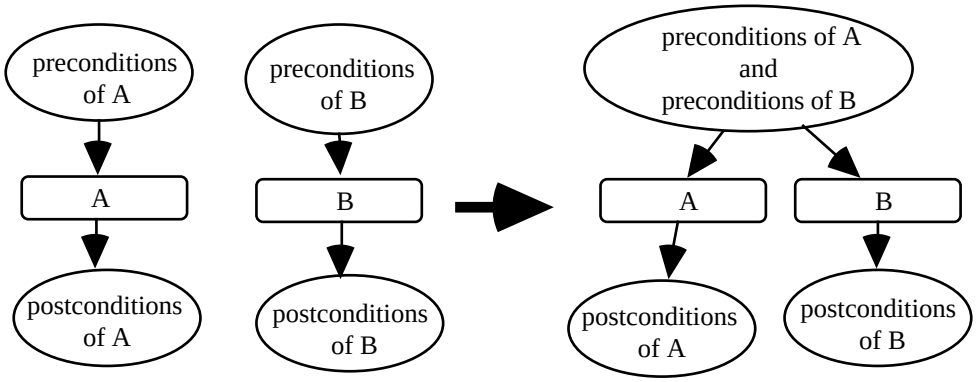


Fig. 3-16 Adding alternatives by merging of states

Furthermore, the composition can be based on the **recognition of some special event** in a MOI that **triggers some behaviour** in the linked ones. This idea was already exploited to make artifacts active (see section 3.3.2) and the notation can be based on the same principles. Then, we can conceive of a *trigger* attribute *for the whole mechanism of interaction* as part of the linking notation: this attribute declares which triggers the mechanism is able to send and react to by activating some internal behaviour (basically, a function).

From all the considerations above, the linking function of the γ-level takes an ***interface*** as an argument in order to define the desired type of linking: in principle it could be the combination of all the possibilities mentioned above. The interface declares which objects are shared, which actions and states have to be merged and finally, which triggers can be sent and received.

The linking notation is specified as a function defined as follows:

Link(g_1, g_2, Interface$_1$, Interface$_2$)
where g_1, g_2 are the grammars generating the C-MOIs to be linked and Interface$_i$ (i = 1, 2) are arguments of the following type:

Interface:
share: <pairs of objects>*;
merge: <pairs of actions>*;
< pairs of states>*;
out-trigger: <triggering-condition,triggering-mode, trigger>*;
in-trigger: <trigger, function>*

where in each pair of objects/actions/states the first component belongs to the related C-MOI while the second is a reference to an object/action/state of the C-MOI to be shared/merged with. For example, in the case of the Bug Report described in Fig. 3.8 where the composition is based on the merging of actions, the mutual reference is expressed by means of the correspondence of a request of information (?) by X to Y with the availability of Y to provide (!) this piece of information to X.

The above piece of notation draws its inspiration from the notion of *encapsulation* (Snyder, 1987) typical of modular languages.

The operational semantics of the compound C-MOI is defined as the combination of the operational semantics of the source C-MOIs under the obligations expressed by the Interface.

As discussed in the introductory example, it might be the case that the linking of C-MOIs also affects the primitives of the resulting C-MOI. In fact, the results of the primitives associated to each of source C-MOI contribute to the definition of the primitives associated to the composed C-MOI. We only illustrate this point through examples, because as of now the notation does not contain a specific syntax for expressing the various possibilities of combining the primitives.

A typical case is the primitive recording the ‘history’. Let again M1 and M2 be two C-MOIs composed into M3. The history of M3 could be the interleaving of the histories of M1 and M2. Let us consider the following scenario: M1 and M2 are two workflows sharing, together with other resources, a classification scheme (called M4). If M1 and M2 are combined into a C-MOI M3, M4 is ‘implicitly’ imported into M3. The combination of M1 and M2 impacts on the history of M4 (about its use, its modifications, and the like), because in M3 two are the workflows contributing to history of M4. This can be easily managed by the primitive building the ‘history’ of M4. As for the history of M3, it could be both the disjoint sum of the two histories, if the user is not interested in their interleaving, or their interleaving, if the user wants to have a unique framework where to reconstruct the behaviour of M1, M2 and perhaps their relationships with M4.

Another case is when the composition is based on the identification of either pairs of states or pairs of actions. Then the resulting C-MOI is activated and simulated adopting the same operational semantics defined for each components.

Linking is of course the most challenging aspect of the proposed notation. The present achievements prove that this idea is feasible under certain conditions while the full development of the related notation and its impacts on the computation complexity needs an additional investigation that will be matter of further research.

3.7. From the notation towards an architecture

We have introduced the concept of MOI as a fundamental means for articulation work and presented a three layered notation for the description of C-MOIs.
We claim that the notation, basically the schema presented in Fig. 3-3 (section 3.2), can be refined and interpreted as an 'architecture' along two axes: vertically, as an architecture supporting the C-MOI life cycle, horizontally, as an architecture for C-MOIs conceived of as embedded systems (Winograd, 1979). This idea is represented in Fig. 3-17.

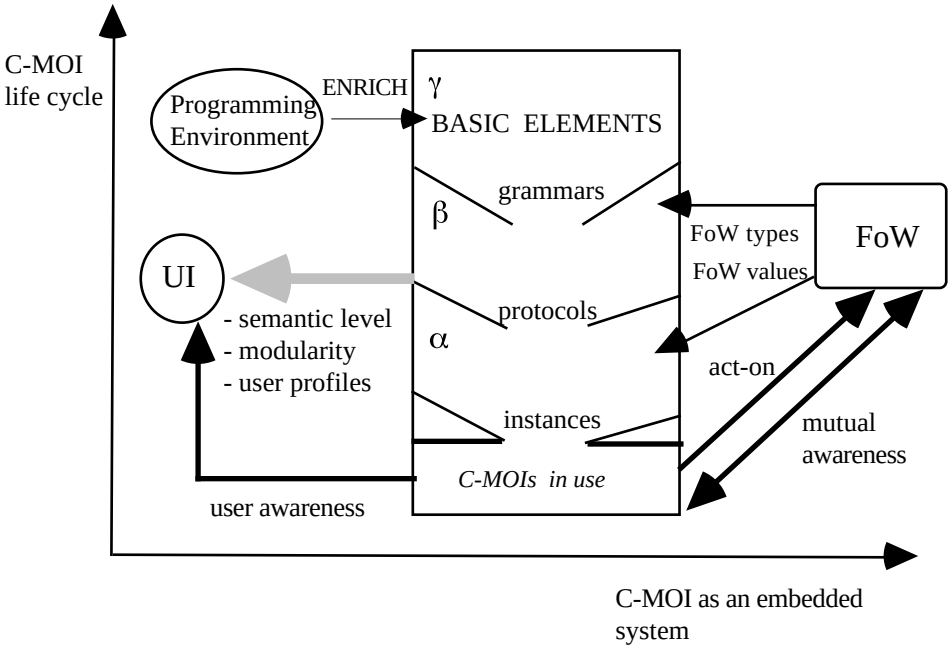


Fig. 3-17. Two complementary views of the architecture underpinned by the notation.

The explanation of these two views allows us to highlight the relationships between the proposed notation and the results achieved by the other Strands of the project, more specifically Strand 2 and Strand 4. The relationship with Strand 1 has been explained in Chapter 1 at the modeling level. It will be matter of the notation too once this latter enters in its implementation phase.

We start by considering C-MOIs as embedded systems. The arrows to the left in Fig. 3-17 connect the C-MOI with the framework in which it is developed: the programming environment and the User Interface. Perhaps this type of embedding goes beyond the classical view of embedding (that corresponds to the connections to the right of the picture). We use this term to stress that the notation is in a dynamic but disciplined relation with its development environment.

The connection with the programming environment is just at the γ-level when new basic elements have to be created or existing ones have to be modified in order to increase the expressive power of the notation in terms of elements or of

their operational semantics. Due to the very nature of the basic elements the platform[1] being under definition in Strand 4 seems to be a programming environment very suitable for the proposed notation. In fact, this platform emphasises both the object oriented view that naturally implements modularity and the awareness at the user level as well as at the level of the objects themselves. This second aspect makes the platform especially useful since each OAW, each active-artifact and compositionally each C-MOI is designed so as to be able to convey information about the changes of its internal states to the environment, and in particular to the UI. The notation has also a logical connection with the UI (represented by the shaded arrow in Fig. 3-17). In fact, the components of the notation, namely objects, structures and the related primitives, provide the UI designer with a modular and compositional 'language' from which different 'phrases' (i.e., UI layouts) can be constructed to meet the users and/or the organization requirements. First of all, the notation provides the UI designer with a set of objects at the appropriate semantic level for which it makes sense to look for an adequate graphical/multimedia representation able to support the same degree of modularity characterising the notation. We see here a strong connection with the work carried out in Strand 4 about virtual reality. Secondly, the visibility of the notation can be tailored to different roles in different application domains. In this way each user can exploit the level of visibility her role is guaranteeing for the design of the most appropriate layout of her working space by exploiting the various primitives and the linking function. The customization of the User Interface according to user's profiles can employ the techniques proposed in the framework of User Modeling (Kobsa and Wahlster, 1989) and in the exploitation of the User Model notion in making CSCW systems adaptive (Divitini and Simone, 1994a).

The arrows to the right connect the C-MOI with the Field of Work, that is it context of use. The meaning of the arrows have been explained in section 3.4 (Fig. 3-11) and section 3.5 (Fig. 3-12) and are recalled here for the sake of completeness. The relevant aspect is that the interface between C-MOIs and their FoW is well defined and provides a guideline on how to fulfill the requirement mentioned in Chapter 1 stating that…

> A computational mechanism of interaction should thus be conceived of as an abstract device incorporated in a particular software application (e.g., a CASE tool, an office information system, a CAD system, a production control system, etc.) so as to support the articulation of the distributed activities of multiple actors with respect to that application — without imposing on actors an undue impedance between articulation work and work.

We can make the conception of C-MOIs as embedded systems more palpable by illustrating the relation between the Notation and one of the platforms proposed in Strand 4 through the diagram of Fig. 3-18 which is derived from Appendix 2 of this deliverable. The Notation has bidirectional relations with the User Interface (SIS), with the various Applications developed over the platform

---

1 We use here the term 'platform' in order to avoid confusing the architecture induced by the notation in Strand 3 with the architecture developed in Strand 4.

(they belong to the Field of Work) and with the shared objects environment (SOS and Comp) which constitutes the programming environment of the Notation.

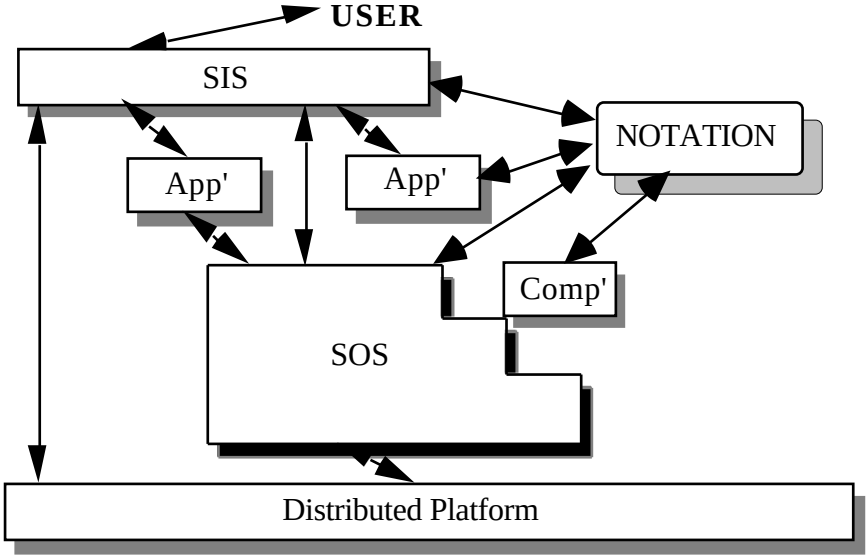


Fig 3-18. The Notation embedded in a Platform proposed in Strand 4.

Let us now consider the view of the notation as an architecture supporting the C-MOI life cycle. In section 3.1 we claimed that the notation should be a means for a smooth transition from the analysis step toward the design step. In this aim, the semantic level, the formal definition and the openness of the notation (see Fig. 3-1) play a basic role since these properties make the notation a language supporting the cooperation of analysts and designers (they need not necessarily to be just specialist but can be also users with different roles and skills, as discussed in section 3.2) during the C-MOI development. Obviously, the analysis of the work setting leading to the identification of the requirements of the needed C-MOI(s) makes use of approaches, methods, languages that go beyond the framework considered here. These topics are the matter of the research conducted in Strand 2 and have been considered in the analysis of MOIs in the field studies reported in the Deliverable 3.2. Once the analysis reaches a point where the requirements have been identified, then the requirement definition step must be initiated. In this activity the notation can be exploited to provide the design step with an unambiguous description of the functionalities the C-MOI has to perform and foremost of how they are allocated to either the users or the computer artifact. This approach can be illustrated in relation to the example developed in Chapter 2 (Fig. 3-19).

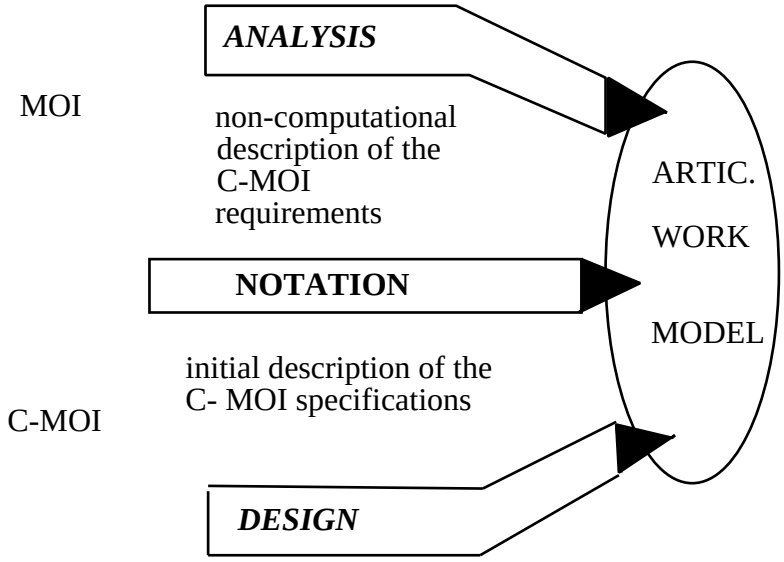


Fig. 3-19. The role of the Notation and of the Articulation Work Model in the C-MOI life cycle in the example of Chapter 2.

The analysis phase ended with a (non-computational) description of the requirements of the Bug Handling process. The requirements are representing a re-engineering of both the artifacts and the related procedures, with a specific attention to the interaction between the users and the desired C-MOI. Now, in order to start with the design, these requirements have to be represented in the notation and developed on the base of the architecture described in the first part of this section. This job is made especially natural by the fact that the Articulation Work Model underlying the notation has been used as one of the conceptual structures used to describe the requirements, and by the fact that the notation possesses adequate means for representing interactions (triggers) from the C-MOI to the UI and vice-versa.

During the design step the initial description is refined by specifying all the needed details as expressed by the notation itself at the various levels. Since this activity is not linear and generally requires to revisit both the requirements and the design choices, the layered and modular structure of the notation and the associated primitives provide an uniform and powerful framework where the revisions can be performed in a controlled and sound way.

This type of support can be provided by the proposed framework also and foremost during the maintenance step that follows the initial implementation. The visibility of the notation makes this activity distributed up to the point where the users are able to locally and or permanently adapt the current computer artifact to their varying needs. However, if the intervention of specialists is needed and some ‘centralised’ development cannot be avoided, the notation plays again the role of communication support among analysts, designers and the end-users.

To sum up, we claim that the notation and the software architecture underpinned by the notation allow for a radical change of the view of system life cycle, in the spirit of the considerations made in Strand 2.

Acknowledgments

The authors want to thank Peter Carstensen, Betty Hewitt, Kjeld Schmidt and Carsten Sørensen for the many discussions that provided them with constructive inputs for the progressive refinement of the notation for computational mechanisms of interaction, and for their detailed revision of the final version of this chapter. All errors in this chapter naturally remain the responsibility of the authors.

References

Agha, Gul and C. Hewitt (1987): Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming. In *Reasearch Directions in Object-Oriented Programming*, ed. B. Shriver and W. Wegner. Cambridge, MA: The MIT Press, pp. 49- 74.

Agostini, Alessandra, Giorgio De Michelis, Stefano Patriarca, and Renata Tinini (1994): A Prototype of an Integrated Coordination Support System. *Computer Supported Cooperative Work (CSCW). An International Journal*, vol. 2, no. 4, pp. 209-238.

Bernardinello, Luca and Fiorella De Cindio (1992): A Survey of Basic Net Models and Modular Net Classes. In *Advances in Petri-nets 92, LNCS 609*, ed. G. Rozenberg. Berlin, Germany: Springer- Verlag, pp. 304-351.

Brauer, W., W. Reisig, and G. Rozenberg, ed. (1987): *Petri-nets: Central Models and Their Properties*, vol. 254. Lecture Notes in Computer Science. Berlin, Germany: Springer-Verlag.

Conklin, J. (1988): gIBIS: A hypertext tool for exploratory policy discussion. In *Proceedings of the Conference on Computer-Supported Cooperative Work (CSCW '88), Portland, Oregon, 26-28 September 1988*. ACM Press, pp. 140-152.

De Cindio, Fiorella , Giorgio De Michelis, Lucia Pomello, and Carla Simone (1982): Superposed Automata Nets. In *Application and Theory of Petri-nets, IFB 52*, ed. G. Girauld and W. Reisig. Berlin, Germany: Springer-Verlag.

Divitini, Monica and Carla Simone (1994a): Adaptivity in a system supporting cooperation. In *Fourth International Conference on User Modeling, Hyannis, MA, USA,15- 19 August, 1994*, ed. B. Goodman, A. Kobsa, and D. Litman. User Modeling Inc., pp. 59-64.

Divitini, Monica and Carla Simone (1994b): A Prototype for Providing Users with the Contexts of Cooperation. In *ECCE7, Bonn, Germany, September 5-8,1994*, ed. R.Oppermann, S.Bagnara, and D.Benyon. GMD, pp. 253-270.

Egger, Edeltraud and Ina Wagner (1993): Negotiating Temporal Orders: The Case of Collaborative Time Management in a Surgery Clinic. *Computer Supported Cooperative Work (CSCW). An International Journal*, vol. 1, no. 4, pp. 255-275.

Ehrig, H., M. Nagl, and G. Rozenberg, ed. (1983): *Graph-Grammars and Their Application to Computer Science (2nd International Workshop)*. Lecture Notes in Computer Science. Berlin: Springer-Verlag.

Ellis, C. and G. Nutt (1980): Office information systems and computer science. *ACM Computing Surveys*, vol. 12, no. 1, pp. 27-60. - Reprinted in Greif, 1988.

Ellis, Clarence A. and Karim Keddara (1994): Dynamic change within workflow systems. Unpublished manuscript, 1994.

Hoare, C.A.R. (1985): *Communicating Sequential Processes*. Series in Computer Science. Englewood Cliff, NJ: Prentice Hall Inc.

Jensen, K. and G. Rozenberg, ed. (1991): *High-Level Petri-nets: Theory and Application*. Berlin: Springer-Verlag.

Johnson, Philip (1992): Supporting Exploratory CSCW with the EGRET Framework. In *CSCW '92. Proceedings of the Conference on Computer-Supported Cooperative Work, Toronto, Canada, October 31 to November 4, 1992*, ed. Jon Turner and Robert Kraut. New York: ACM Press, pp. 298-305.

Kobsa, A. and W. Wahlster, ed. (1989): *User models in dialog systems*. Berlin, Heidelberg, Germany: Springer-Verlag.

Kreifelts, T. and G. Woetzel (1987): Distribution and Error Handling in an Office Procedure System. In *Office Systems: Methods and Tools*, ed. G. Bracchi and D. Tsichritzis. North-Holland: Elsevier Science Publishers B.V., pp. 197-208.

Medina-Mora, Raul, Terry Winograd, Rodrigo Flores, and Fernando Flores (1992): The Action Workflow Approach to Workflow Management Technology. In *CSCW '92. Proceedings of the Conference on Computer-Supported Cooperative Work, Toronto, Canada, October 31 to November 4, 1992*, ed. Jon Turner and Robert Kraut. New York: ACM Press, pp. 281-288.

Milner, R., J. Parrow, and D. Walker (1992): A Calculus of Mobile Processes, I-II. *Information and Computation*, vol. 100, no. 1, pp. 1-40, 41-77.

Nilsson, N.J. (1980): *Principles of Artificial Intelligence*. Palo Alto, CA: Tioga Publishing Co.

Pomello, Lucia, Grzegorz Rozenberg, and Carla Simone (1992): A Survey of Equivalence Notions for Petri-net based Systems. In *Advances in Petri-nets 92, LNCS 609*, ed. G. Rozenberg. Berlin: Springer- Verlag, pp. 410-472.

Reisig, W. (1985): *Petri-nets: an Introduction*. Berlin, Germany: Springer- Verlag.

Rodden, Tom, John A. Mariani, and Gordon Blair (1992): Supporting Cooperative Applications. *Computer Supported Cooperative Work (CSCW). An International Journal*, vol. 1, no. 1-2, pp. 41-68.

Searle, John R. (1975): A Taxonomy of Illocutionary Acts. In *Language, Mind, and Knowldge*, ed. Keith Gunderson, vol. VII. Minneapolis: University of Minnesota Press, pp. 344-369.

Simone, Carla and Kjeld Schmidt, ed. (1993): *Computational Mechanisms of Interaction for CSCW*. COMIC, Esprit Basic Research Project 6225. Lancaster, U.K.: Computing Department, Lancaster University. - [COMIC Deliverable 3.1. Available via anonymous FTP from ftp.comp.lancs.ac.uk].

Snyder, Alan (1987): Inheritance and the Development of Encapsulated Software Systems. In *Research Directions in Object-Oriented Programming*, ed. B. Shriver and P. Wegner. Cambridge, MA: The MIT Press, pp. 165- 188.

van Leeuwen, J. (1990): Graph algorithms. In *Handbook of Theoretical Computer Science*, ed. J. van Leeuwen, vol. A, algorithms and complexity. The Netherlands: Elsevier Science Publishers B.V., pp. 525-632.

Winograd, Terry (1979): Beyond Programming Languages. *Communications of the ACM*, vol. 22, no. 7, pp. 391-401.

Winograd, Terry and Fernando Flores (1986): *Understanding Computers and Cognition: A New Foundation for Design*. Norwood, New Jersey: Ablex Publishing Corp.

Appendices

On adaptable support for cooperative work

Mads Dam

Swedish Institute of Computer Science

We argue that a critical dimension in the handling of change in computer based systems for cooperative work is whether change should be explicitly embedded into systems, or whether change should be handled in a global and uniform manner, for instance by a process of editing and recompiling programs or scripts on the fly. We propose a formal basis for the description of dynamically modifiable objects, and explore its applicability in the field of CSCW by exposing it to three examples of increasing complexity: A system for dynamic communication channel creation; an adaptable conversation manager; and a rudimentary , yet quite general, awareness model.

1. Introduction

Cooperative work is a distributed activity. Actors engaged in cooperative work have differing tasks, knowledge, responsibilities, authorities, obligations with respect to the work at hand. For the distributed activities to result in any useful outcome individual actors tasks needs to be coordinated, knowledge to be shared, responsibilities and authorities to be delegated, obligations to be inherited, and so on. *Articulation* is this process of controlling and coordinating distributed activities. In general, articulation is overhead, and a major yardstick in measuring the quality or efficiency of systems for supporting cooperative work is the extent to which this overhead is minimised.

1.1. The Articulation of Change

Supporting articulation, however, is hitting a moving target. Actors, tasks, and organisations change, and the requirements for articulation support changes with them. Making systems adapt to and reflect this change is one of the serious challenges in CSCW systems design. For the purpose of the present discussion let us by a *device,* or *mechanism of interaction* mean a computational artifact intended to support the articulation of cooperative work in some given application. In attempting to make devices adaptable to change (at least) the following issues needs to be addressed:

1. What changes are considered? Are all changes to be handled on an equal footing?

2. Should facilities for change be built into applications, or should change be handled ``from above'', for instance by editing source code or program scripts?
3. Who should be empowered to perform modifications (transient, or permanent) to what parts of the device?

Of course, in prototype applications with just a few participants informed in advance of the architecture and workings of a given device these issues may be largely irrelevant. It is the system designers themselves who cooperate, they are familiar with the programming and script languages concerned, they are familiar with the workings of the devices, and they act in a ``friendly'' manner. This situation is, however, in marked contrast to the much more demanding regime of large-scale use in modern industrial and administrative organisations. Such organisations are *structured* (different parts of the organisation have different tasks and responsibilities), *heterogenous* (different parts of the organisation have different and possibly diverging views of the organisation as a whole, they are unequally informed, they have different aims and aspirations), and *temporal* (structure and participants change over time). For such organisations we argue that the nature of changes become of the utmost importance, and that qualitatively new distinctions arise. There is, for instance, a qualitative distinction between *A* taking over the responsibilities of *B* while *B* is on holiday, and of modifying the inner workings of a protocol used to delegate responsibilities to third parties. As responsibilities are delegated and subdivided, so, typically, are powers of change. Moreover there is no reason a priori to believe that these subdivisions necessarily follow simple syntactic or structural patterns. Rather, the process of negotiating, delineating, and delegating powers of change is, we believe, the central aspect of articulation that supports its dynamic nature. Notice that by this we have stated that articulation of change is itself articulation, and as such it is itself of a local and temporary nature.

1.2. External Handling of Change

This position, then, is at odds with an approach which lifts devices for articulating change out of applications to treat them in a static, global and unstructured manner. One straightforward way of following such an approach is to furnish end-users with unlimited and unrestrained visibility and power of modification of programs, scripts, programming languages, and devices. Several recent suggestions for building adaptability and flexibility into CSCW devices, however, appear to advocate doing just this (cf. Malone et al, 1992; Kaplan et al, 1992; Simone et al, 1994; Schmidt et al, 1993).

For instance, in the approach of (Simone et al, 1993) a three-level language is introduced which on its lowest level has device instances, on its second level device specifications, and on its third level the device specification meta-language. Devices are modified, transiently or permanently, in effect by editing

the textual representations of the device specifications, allowing some unspecified extent of modification to the device specification meta-language, should this be deemed to be necessary. Moreover, at the device instance (``run time'') level, change articulation is handled in a manner which is entirely independent of the application at hand, by providing uniform, unstructured and global powers of change.

This failure to specify how, when, why, to what extent, and by whom changes may be articulated has serious consequences:

1. Articulation is not to be trusted. This follows, since powers of change are global and unrestrained, and since participants have divergent views, responsibilities, aims and aspirations concerning the tasks at hand. As a consequence the entire cooperative process cannot be trusted either to deliver any useful outcome, or in the extreme not to jeopardize the whole existence of the organisation by for instance destroying, modifying, or revealing crucial information.
2. Even in ``friendly'' environments, articulation will fail to live up to its requirements. Articulation work can be extremely subtle. Consider, for instance, the problem of guaranteeing actors mutually exclusive right of access in a pure shared information setting. The difficulties that computer science encountered in devising correct solutions to this problem are well-known to any computer science undergraduate. Now, it may be argued that problems such as mutual exclusion are much too low-level and primitive, and that problems of articulating cooperative work are at a far higher level of abstraction. While this may or may not be true, it is at any rate clear that devices in real organisations can be extremely subtle and that changes (due, for instance, to well-meaning but uninformed action) can have highly detrimental or disastrous results (due, for instance, to information loss).
3. Articulation overhead increases rather than decreases. Giving end-users unrestrained and global powers of change means that these powers will also be used. This follows from the heterogenous and temporal nature of organisations. Changes will sometimes be detrimental to the way devices support the processes they were intended for, and sometimes they will not. But, being global and unstructured, each change will have the potential to affect the articulation support of large numbers of actors. Those actors will have to understand, be informed, and relate to these changes, and their own expectations and usage of device behaviour must change. And, indeed, they will in general have to take corrective action, since participants are not always informed or friendly. Unless, of course, changes are made in a collaborative fashion, hoping that because of this eventually the desire for change will subside. But this contradicts the initial assertion (that powers of change will indeed be used).

1.3. Embedded Handling of Change

The conclusion, then, is that powers of change must be local, structured, and dynamic, and supported as an integral aspect of articulation mechanisms in general. Indeed this is exactly what happens in most ``real-world'' situations. A prime example is the legal hierarchy. But there are lots of examples on smaller scales:

* Companies have devices handling the situation of employees going on vacation.
* Assemblies have devices for members taking the word out of turn.
* Control rooms have devices for handling exceptional situations. Such as: Talking out loud to open a channel of communication to your supervisors and peers (cf. Heath, Luff, 1992), or ``cocking out'' flight strips to point attention to potential future conflicts in an air traffic control room (cf. Schmidt, 1993).
* Token passing manufacturing systems such as the kanban systems considered by Schmidt (Schmidt, 1993) have devices for changing their configuration (e.g. by pocketing or reinserting kanban cards).

Certainly these articulation change devices are not always considered part of the ``normal'' operation of the larger device they are meant to support (i.e. kanban systems, assemblies). Nonetheless they do have an explicit existence, and have rules for their operation honed by use. The point is that these types of change are handled, intentionally or not, ``from within'' the device rather than ``from without'', and that the powers of change far from being unstructured and global are localised and structured. For instance, in a kanban system, changes such as removing a kanban card should be contrasted with, say, suddenly introducing colour coding of kanban cards and instructing operators in their proper interpretation.

Notice that this point does not mean that we oppose end-user programming in general, but rather that the capabilities of end-user programming should be carefully controlled to ensure that types and powers of change are respected. Neither does it mean that we advocate a procedural approach over one based on information sharing. This issue is entirely orthogonal to the one we consider here. Nor does it mean that we believe all aspects of change can or should be foreseen. Devices and their usages are honed by practice. In some cases devices already have the features needed to reflect this process. In other cases there will be a need to introduce new devices and scrap old ones by global and unrestrained change. Maybe this process can even be supported by communities of researchers and developers cooperatively editing and defining their programs, scripts, metatheories, and so forth.

1.4 Notations for Change

The problem, however, is that even if it is granted that devices should support their own reconfiguration in a dynamic, local, and structured fashion, it is far from clear how this goal is to be achieved. Dynamically reconfigurable systems can be extremely subtle and we currently have very few tools that describe systems with such facilities in concise, intuitive, and tractable manners. In the field of concurrency, however, dynamically reconfigurable systems have recently been the subject of considerable attention with the advent of the pi-calculus of Milner, Parrow, and Walker (Milner et al, 1992) and the discovery of the remarkable usefulness and power of this calculus (cf. Milner, 1991; Sangiorgi, 1993; Orava and Parrow, 1992). The purpose of the present paper is to investigate the extent to which concepts introduced with the pi-calculus can be used as a basis for deriving notations useful for describing dynamically evolving systems such as those encountered in CSCW. Our contribution should be viewed as a first step in a largely experiment-driven sequence of refinements resulting from attempts to specify and analyse ever larger and more complex devices. We offer a small statically typed calculus of objects based on the pi-calculus and give example specifications of user reconfigurable devices that illustrate the use of this calculus.

Object-based notations have been used in other contexts for describing user modifiable systems for cooperative work. Oval (Malone et al, 1992) for instance, is a system for implementing devices which can be modified by for example adding, changing, or removing subtypes, fields, or views of objects. Other related approaches include the schemas and schema instances of EGRET (Johnson, 1992) and the communication structures of COSMOS (Young, 1989). Compared to these approaches ours is much more primitive. Users are not provided with any facilities whatsoever for accessing, modifying or editing source code. Rather, the only interaction primitives provided are

1. the passing of vectors of channel names along a channel, and
2. the generation of new local channel names.

By working in a statically typed setting channel names can be viewed as object instances, and synchronising on a channel can be viewed as accessing an object. By generating new object instances and passing them around it becomes possible to govern the way object instances can access each other in a dynamic way. This is the basic mechanism for modelling the dynamic modifiability of interconnection topology or information visibility. What may be surprising is that even though no CSCW-specific distinctions (in the sense of Flores et al, 1988) are built into our calculus the few primitives that are provided are nonetheless sufficient for describing central concepts in CSCW such as dynamic conversation management or the awareness models of (Benford and Fahlén, 1993), the examples studied in the present note. These examples have been chosen to illustrate two orthogonal types of device of relevance to CSCW.

First, the conversation manager provides an example of a device for dynamically modifying articulation procedures. Conversation systems such as

Conversation-for-Action (Flores et al, 1988) are nothing but state transition systems with transitions labelled by so-called illocutionary points. It is natural to identify illocutionary points with channel names. Thus conversation systems have very natural representations in our calculus. Conversation systems can conceivably be subjected to modification in a variety of ways. Here, as an example, we consider supporting the dynamic addition and removal of states, illocutionary points, and transitions.

Secondly, the awareness model (Benford and Fahlén, 1993) provides an example of a device for dynamically modifying information visibility. This model has been developed to support navigation in large shared data spaces. Its key feature is the support of dynamically modifiable levels of awareness between objects. It is an excellent illustration of the main tenet of the note, namely the virtues of building support of change into devices rather than leaving it to be handled at the meta-level. The basic device of the awareness model is *n*-dimensional space (typically, n =3). The device is structured by, for instance, populating space by objects of different types (representing users, other devices, or autonomous agents), or by bounding space in different dimensions. Objects communicate through the exchange of information such as position, text, structured data, video- or audio streams. The process of populating and bounding space is the static support of articulation provided by the device. In large spaces, however, this support is wholly inadequate, and some dynamic control of object ``visibility'' (in a general sense) is needed to avoid computational and perceptory overload. The awareness model articulates this change by determining levels of awareness (e.g. visibility) according to the distance between objects, and it is just from this simple and intuitive ``internal'' support of articulation change that the awareness model derives its strength.

The note is structured as follows. We start by introducing the basic specification language and notions of type, object, and object reference by means of a few simple examples related to CSCW. Then we illustrate the application of this language by first considering a simple adaptable conversion manager, or programmable finite state automaton. As a second example we consider a core awareness manager based on the ideas of Benford and Fahlén. The example refines the awareness model somewhat by introducing distinctions between emissive and receptive awareness hierarchies, and by eliminating the concept of aura. Finally in the conclusion we briefly discuss the prospect of applying analysis tools to the current specification language.

2. A Typed Calculus of Concurrent Objects

We introduce a simple calculus that talk about object types, objects, object references, and channels. Our intention is to describe *objects* as concurrent processes. Objects possess *channels* along which they interact with other objects by synchronisation and, potentially, passing of *object references* . Objects may

interact with other objects through its own channels or through the channels of other objects known to it. *Object types* describe what channels are possessed by its objects, their directionality, and the types of object references passed.

2.1 Types

The type of an object reference is the type of object it references. For instance, `SEMAPHORE` is the type of semaphores possessing an outgoing channel `wait` (or, often, *P*), and an ingoing channel `release` (or *V*), along which nothing is passed:

```
type SEMAPHORE =
 channels
 wait!,
 release?
 end;
```

Pictorially we can represent an object of type `SEMAPHORE` as in fig. 1.

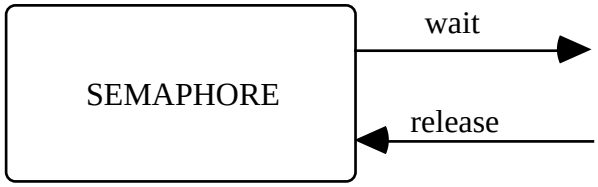


Figure 1: Semaphore object

Any object *A* that knows about a `SEMAPHORE` object *S* has the potentiality of synchronising on the channels `wait` or `release` possessed by *S* , provided that *S* chooses to offer such synchronisations. Note that passing nothing is not the same as passing an object reference of the unit type `()`: Objects of the unit type possess no channels of their own, but they may still be able to interact with their environments by knowing about other objects. We allow recursive objects. Here, for instance, is the type of natural number objects:

```
type NAT =
 channels
 zero!,
 succ!NAT
```

```
end;
```

Thus, an object of type `NAT` has two outgoing channels, one along which nothing is passed, and one along which a reference to an object of type `NAT` is passed. We also allow polymorphic channels such as the type of buffers of objects of type `a'`[1]:

```
type a' BUFFER =
 channels
 in?'a,
 out!'a
 end;
```

In addition to these basic type constructing mechanisms only a few more are used: A mechanisms for sequentialising channels, and, more importantly, a kind of product type used for providing non-uniform *views* of objects. These are described as they are introduced in the examples that follow.

2.2 Objects and References

Types only determine the *static* structure of objects. The way objects actually use the channels they themselves possess as well as what environments objects require to function correctly, and the way these environments are used, is determined by the *dynamic behaviour* of objects. In its simplest form an object is just a process together with a reference to it through which the process can be queried for the identity of the channels it possesses. Channel identities, or *names*, are subject to dynamical change, and it is the "owning process'' that controls this change. Typically (though this feature is by no means essential), with the type system as described so far, channel names get outdated as soon as they are used. Object references, however, are static, as are the types of objects they reference (though the objects themselves, of course, are subject to change). We use the notation `Obj:OBJ-TYPE @ obj-ref` for an object `Obj` of type `OBJ-TYPE` referenced by `obj-ref`. The usage of capital letters is deliberate to maintain a syntactical distinction between types, objects, and references. If the type of `Obj` is well-determined then `Obj:OBJ-TYPE @ obj-ref` can be abbreviated `Obj @ obj-ref`, and similarly `OBJ-TYPE @ obj-ref` is used when `Obj` is indeterminate.

As a first example we consider a polymorphic one-element buffer of type `'a BUFFER`. Such an object alternates between an empty and a full state. The behaviour of objects in these states is described by mutual recursion as follows:

---

1 Following the ML tradition (c.f. Paulson, 1991) we use 'a, 'b, etc as type variables.

```
object
 EmptyBuf: ‘a BUFFER @ b =
 in?x.FullBuf(x) @ b
and
 FullBuf(‘a @ x):’a BUFFER @ b =
 out!x.EmptyBuf @ b
```

The operational intuition is as follows: At the initial state `EmptyBuf` is an object of type `‘a BUFFER` referenced by `b` . The only thing that can be done to `EmptyBuf` is referencing it since the channels owned by `EmptyBuf,` that is, `in` and `out`, have not yet been instantiated. Referencing `EmptyBuf` causes `in` and `out` to be instantiated. `EmptyBuf` then offers the input of an object reference `x` of type `‘a` along the current instance of `in` to whoever knows about this current instance, namely the object that referenced `EmptyBuf` in the first place. No other interactions are offered by `EmptyBuf` at this stage. Then, after having input such an `x` , `EmptyBuf` turns into an object `FullBuf` parametrised on `x` , and referenced by `b` . Moreover, the instances of `in` and `out` created by referencing `EmptyBuf` now becomes void, and a new reference-use cycle is needed for an object to interact with the object referenced by `b` .

This process of referencing and instantiating will be well-known to those familiar with the pi-calculus. It is the same mechanism that underpins, for instance, the encoding of data types in the polyadic version of the pi-calculus of (Milner, 1991). Object references are nothing but channels, and instantiating, e.g., `in` and `out` by referencing `b` in the example above amounts to passing from `EmptyBuf` along `b` a new unique pair of channel names. Thus in and out in the definition of `EmptyBuf` above should not be viewed as global channel identifiers, but rather as place-holders, or private names, that become instantiated according to the type description concerned.

It may be a convenience to allow this instantiation process to stretch over more than one local channel use. For instance we may like to code the one-element buffer instead as

```
object
 Buf:’a FANCIER-BUFFER @ b =
 in?x.out!x.Buf @ b
```

This example illustrates the use of prefixing, as in `in?x.out!x.Buf @ b`, to represent the sequencing of actions `in?x` and `out!x`. Note that we can not take `FANCIER-BUFFER = BUFFER,` since we interpret the definition of `BUFFER` as stating that the lifetime (in number of interactions) of `in` and `out` is at most 1. To compensate for this we can allow ε-free regular expressions such as

`in?'a.out!'a` in place of just `in?'a` or `out!'a` in the definition of `'a BUFFER` to allow `Buf` to be typed by the following type:

```
type 'a FANCIER-BUFFER =
 channels
 in?'a.out!'a
 end
```

It is important to bear in mind, however, that `EmptyBuf` is more generally applicable than `Buf`. This is essentially because the reference to `EmptyBuf` can be distributed to several objects such that one object can be responsible for input and another for output. This can not be done with `Buf`.

Using `Buf` we can now give a very simple model of an exclaimer object notifying an observer object to create a new communication channel along which information can subsequently be passed. The structure of the system is shown on fig. 2.

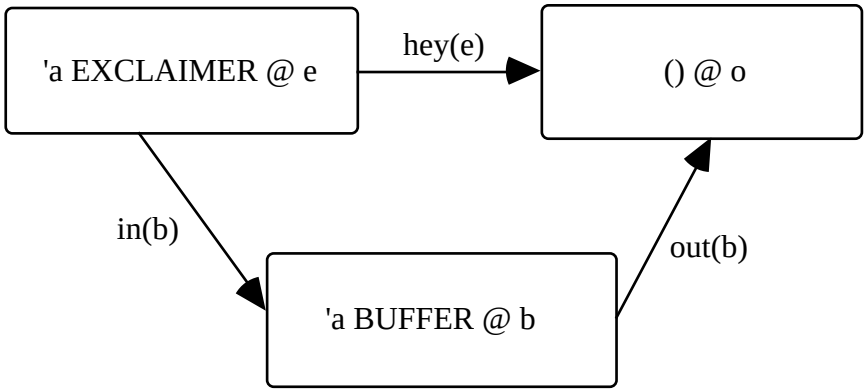


Figure 2: Channel creation system .

The type of exclaimer objects is

```
type 'a EXCLAIMER =
 channels
 hey!'a BUFFER
 end;
```

An exclaimer object that exclaims `hey` and then proceeds to repeatedly output a reference to an `'a` object along a newly created buffer can be described as follows:

```
object
 Exclaimer('a @ x):'a EXCLAIMER @ e =
 new 'a BUFFER @ b
 in hey!b.(Speaker(x,b) @ e | EmptyBuf @ b) end
and
 Speaker('a @ x,'a BUFFER @ b):'a EXCLAIMER @ e =
 in(b)!x.Speaker(x,b) @ e;
```

This example illustrate the use of the channel declaration (or in CCS terms: *restriction*) operator `new` that in this case declares a new unique object reference `b` of type 'a BUFFER. An object that observes an exclaimer to receive a buffer reference along `hey` and then proceeds to listen to whatever the exclaimer chooses to pass along the buffer created can be described as follows:

```
object
 AlertObserver('a EXCLAIMER @ e) @ o =
 hey(e)?b.ActiveObserver(b) @ o
and
 ActiveObserver('a BUFFER @ b) @ o =
 out(b)?x.ActiveObserver(b) @ o;
```

Assuming a primitive type `INFORMATION` a full system composed of an exclaimer and an alert observer is described as the object expression

```
new INFORMATION @ i;
 INFORMATION EXCLAIMER @ e;
 () @ o
in (Exclaimer(i) @ e | AlertObserver(e) @ o) end;
```

It is an easy exercise to modify this system for instance to let exclaimer create not only the buffer used for conversation but also another object used for signalling end-of-conversation.

In addition to the operators introduced in the above examples only a few more operators are used, notably the sum operator + to accommodate alternative behaviours, and the conditional `if - then - else -` .

3. An Adaptable Conversation Manager

In this section we provide a rudimentary example of a type of adaptable device suitable for procedural aspects of articulation work. The example offered is based on the conversation model of Winograd and Flores, considered also by Simone et al (1994). This model provides suitable example material since it is both well known and quite simple.

Essentially, a conversation manager is just a finite state automaton with transitions labelled by so-called illocutionary points, or actions. A practical conversation manager would include various other machinery, derived from this basic structure, such as knowing about user identities, managing multiple simultaneous conversations, keeping agendas of user states and roles, and so forth. A basic method for making conversation managers adaptable and generic is to include support for adding or removing actions, and for adding or removing transitions. These are the types of modification considered in the present example. Other, more structured, approaches to change are conceivable such as allowing conversation structures to be *composed* in various ways out of more primitive ones, for instance by operators such as those of CCS or CSP.

Basic to the adaptable conversation manager are the types of states and actions:

```
type STATE = ();
type ACTION = channels act? end;
```

States have no channels attached to them: They are just references that are passed around between other objects and compared for their identity. Actions, on the other hand, possess an input channel which is used for engaging in the associated speech act. New states or actions are created by declaring references to them. They are considered removed when no more references to them exist. An alternative is to consider explicit remove operations. This could be implemented by introducing additional data structures to keep track of what objects have been created and where they are used.

The finite state automaton is implemented by a central *control memory* responsible for maintaining the current state and for generating transitions, together with a number of *transition objects*. Transition objects compete for access to the control memory. Transition objects possess two channels: One channel for its associated action, and one channel through which the transition object can be prompted for its removal. As a first approximation a suitable type for transition objects is thus

```
type TRANSITION =
    channels
      rm?
      do?
```

```
end;
```

In general, however, it is probably not suitable to have removal of actions being free for all. Rather, we would like access to objects such as `TRANSITION` being controllable such that right to engaging in an action not necessarily implies right of modification. In the current version of the calculus the only way of supporting such distinctions is to distribute a `TRANSITION` object between a remove and an action object. We view this as not reflecting properly the intuitive structure of the system, and, as a more technical point, it introduces additional synchronisation overhead. An alternative is to consider references as vectors, each vector entry being responsible for one ``view'' of the entire object. For this purpose we introduce a product type construction `TYPE-A x TYPE-B` with the operational intuition that a reference to an object of type `TYPE-A x TYPE-B` is a pair of references `(r1,r2)` such that `r1` has type `TYPE-A` and `r2` has type `TYPE-B`. Notice that the product type constructor is quite different from the usual set-theoretic product consisting set of pairs due to the persistent nature of objects. Using products knowledge of an object of type `TYPE-A x TYPE-B` can be distributed by distributing knowledge of its component parts in different ways. For the `TRANSITION` object we introduce a type for the control aspect of transition objects and then define

```
type CONTROL = channels rm? end;
type TRANSITION = CONTROL x ACTION;
```

We consider a `TRANSITION` object as having two ``views'', ``aspects'', or ``roles'', a `CONTROL` aspect for removing transitions, and an `ACTION` aspect for the actual firing of transitions. Finally, the control memory has type

```
type CONTROL-MEMORY =
    channels
          current!STATE,
          next?STATE,
 new-transition?(STATExACTIONxSTATE),
 get-transition!CONTROL
 end;
```

Here `current` and `next` are the channels devoted to querying about and updating the current state, `new-transition` is the channel responsible for receiving transition creation instructions, and `get-transition` is the channel which upon creation of a new transition is responsible for returning a reference to its control aspect. This completes the description of the static structure of the programmable automaton.

For the description of the dynamic semantics we need to describe the behaviour of transitions and control memories. First, to control the distributed access to the control memory we use a semaphore. Semaphores are just simple one-element buffers:

```
object
    Semaphore:SEMAPHORE @ s =
      wait!.ClosedSemaphore @ s
and
    ClosedSemaphore:SEMAPHORE @ s =
      release?.Semaphore @ s;
```

Transitions, now, are objects referenced by pairs of references, the first element of which references the control aspect of transitions, while the second references the action aspect:

```
object
    Trans(CONTROL-MEMORY @ m,
 SEMAPHORE @ s,STATE @ b,
 STATE @ e): TRANSITION @ (c,a) =
      rm(c)?.Idle @ (c,a)
          +
      wait(s)?.
      current(m)?b1.
      if b1 = b
      then
 act(a)?.next(m)!e.release(s)!.
 Trans(m,s,b,e) @ (c,a)
 +
       release(s)!.Trans(m,s,b,e) @ (c,a)
      else
 release(s)!.Trans(m,s,b,e) @ (c,a);
```

Transition objects will, until they are requested to die, continually poll the control memory to see if they are enabled, and the environment, through their action references, to see if they should be fired. Thus, in its initial state, a `Trans(m,s,b,e)` object has two options: It can either be told to die by being accessed along the `rm` channel referenced by `c`, or else it will ask the semaphore for access to the control memory. Having been granted access the current state is read and it is checked if the current state is the origin state of the transition. If it is not the semaphore is released and the transition returns to its initial state. If it is

two options will be open. Either an `act` action is offered by the environment and taken by the transition whereupon the control memory is updated and the semaphore released or else the transition backs off and spontaneously releases the semaphore.

Note the polymorphic constant `Idle`. In implementation terms the object `Idle` of type `CONTROL x ACTION` referenced by `(c,a)` will continually if referenced by e.g. `c` supply an instantiation of `rm` without, however, at any time actually offering a `rm` interaction.

Finally we describe the dynamic semantics of control memories:

```
object
 ControlMemory(STATE @ v,
 SEMAPHORE @ s):CONTROL-MEMORY @ m =
 current!v.ControlMemory(v,s) @ m +
 next?v1.ControlMemory(v1,s) @ m +
 new-transition?(b,a,e).GenTrans(v,s,b,a,e) @ m;
and
 GenTrans(STATE @ v,
 SEMAPHORE @ s,
 STATE @ b,
 ACTION @ a,
 STATE @ e): CONTROL-MEMORY @ m =
    new CONTROL @ c
    in
      get-transition!c.
      (ControlMemory(v,s) @ m |
 Trans(m,s,b,e) @ (c,a))
          end;
```

So a control memory reports its current state, allows the state to be updated, and receives requests for the creation of new transitions. Upon receiving such a request a new control reference is created. The action reference is shared between all transitions labelled by that action, so new action references should not be created. Then a new transition object with origin state `b`, target state `e`, and action reference `a` is spawned off. It is important to note that, since the action reference is shared it will not in general be the case that references determine their corresponding objects uniquely. Several `ACTION` objects may coexist all referenced by the same reference `a`. The offer of an action `act(a)` may be responded on by any of these objects in a non-deterministic fashion. All that is known at run-time is that the reference is type-safe. In this sense the term

``reference'' may well be regarded as a misnomer and should in the future be replaced by a more appropriate term.

The adaptable conversation manager in its initial state is then described by the following composite object:

```
object ConversationManager(STATE @ initial):
 CONTROL-MEMORY @ m =
 new SEMAPHORE @ s;
      STATE @ initial
 in Semaphore @ s | ControlMemory(initial,s) @ m
 end;
```

To give an example run of the conversation manager we demonstrate one way of generating the simple repetitive conversation-for-action structure shown on fig. 3:

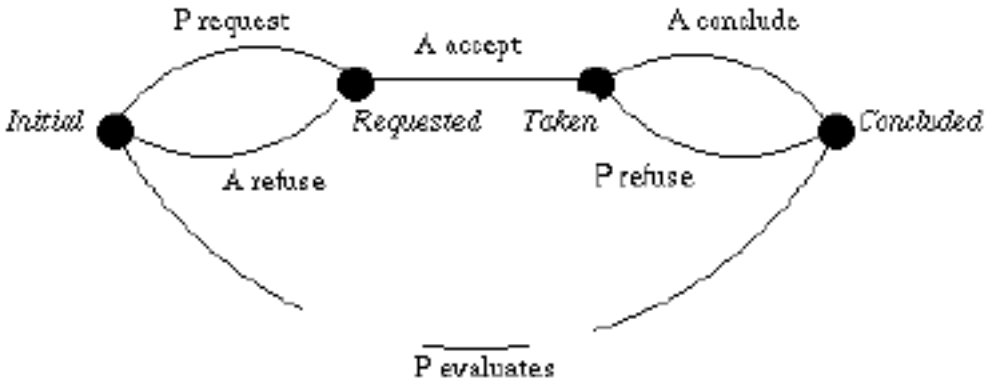


Figure 3: Example conversation-for-action structure

```
system SimpleCfA(ACTION @ p-request,
                 ACTION @ a-accept,
                 ACTION @ a-refuse;
                 ACTION @ a-conclude;
                 ACTION @ p-refuse;
                 ACTION @ p-evaluate) =
new STATE @ initial;
 STATE @ requested;
 STATE @ taken;
```

```
    STATE @ concluded;
    CONTROL-MEMORY @ m
in
 ConversationManager(initial) @ m
      |
 new-transition(m)!(initial,p-request,requested).
 get-transition(m)?_.
 new-transition(m)!(requested,a-refuse,initial).
 get-transition(m)?_.
 new-transition(m)!(requested,a-accept,taken).
 get-transition(m)?_.
 new-transition(m)!(taken,a-conclude,concluded).
 get-transition(m)?_.
 new-transition(m)!(concluded,p-refuse,taken).
 get-transition(m)?_.
 new-transition(m)!(concluded,p-evaluates,initial).
 get-transition(m)?_.
    Nil
end
```

Here `Nil` is the inaction constant of CCS, and _ is used as a wild-card character. Since all object references except those of actions have been internalised in this system, it does not strictly speaking represent an object. Thus no type or reference is assigned to it. Alternatively, it could be assigned the trivial type () and an arbitrary reference. Indeed system declarations can be viewed as abbreviations for such kinds of objects. As another alternative it would have been entirely possible to let the reference `m` to the conversation manager also serve as reference to `SimpleCfA`, thus having `SimpleCfA` to be an object of type `CONTROL-MEMORY`. We choose not to do this in this example in order to illustrate the situation when no such references remain and thus to the outside observer the system has been brought into a state from which it can not in fact be structurally modified.

4. An Awareness Manager

The awareness model of Benford and Fahlén (1993) was introduced as a device for dynamically modifying information access capabilities among users and resources in large shared data spaces. As the number of users or devices in such spaces gets large, maintaining full visibility between objects gets both computationally and cognitively infeasible. The awareness model addresses this problem by imposing a metric onto data spaces, using the resulting notion of

distance to govern the awareness between objects, viz. the amount of filtering that information goes through in passing from one object to another. Note that awareness is fundamentally a direction- and media-dependent notion. The awareness that *A* has of *B* has in general no bearing on the awareness *B* has of *A*. Neither need there be correlation between e.g. visual and aural awareness. The concepts of *focus* and *nimbus* have been introduced to capture the directionality of awareness. Focus determines the *receptive* aspects of awareness, and nimbus the *emissive* aspects. The awareness *A* has of *B* (along a given medium) is then determined by *A*'s focus upon *B* together with *B*'s nimbus upon *A*.

The notion of *aura* has been proposed as a basic mechanism for channel creation and removal, and as enabler of the mechanisms that in turn support focus and nimbus. Informally, an *aura* is a subspace attached to an object, and communication between objects is enabled when their auras intersect, or *collide*. The basic idea is illustrated on fig. 4.

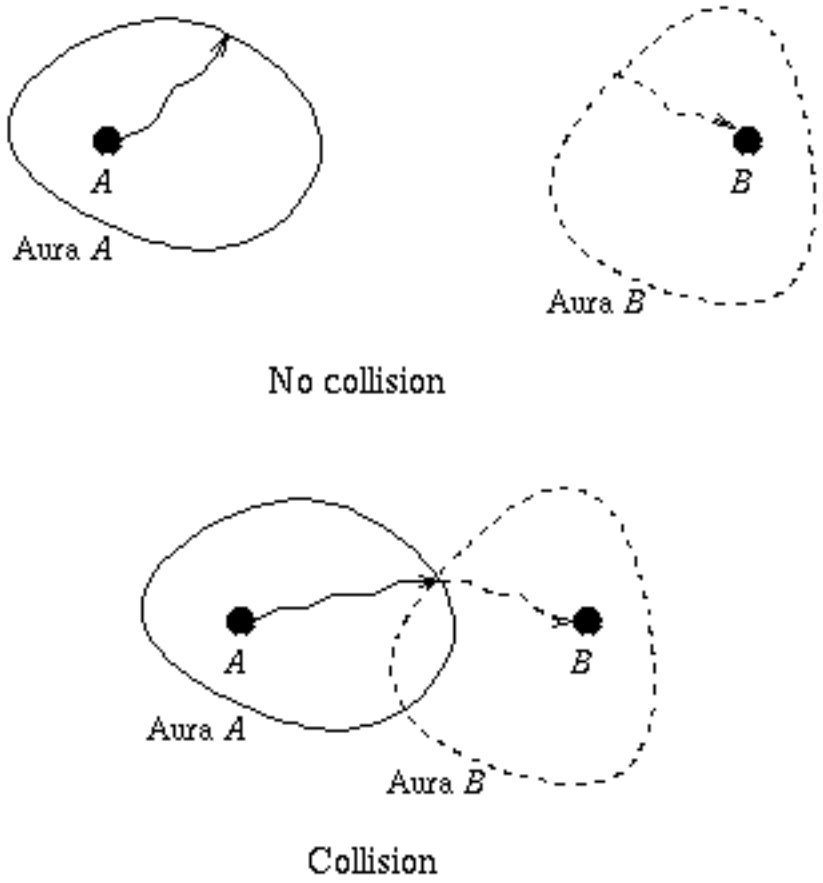


Figure 4: Basic aura mechanism

Objects *A* and *B* are shown as points surrounded by their associated auras. Objects move around in space, translationally or rotationally. Object movements are mimicked by their associated auras. When auras do not intersect no

communication takes place (along the particular medium considered). When an aura collision takes place a channel of communication is created for passing (in this case) information from *A* to *B*. Conversely, when a state of collision ceases to hold this communication channel is removed.

This basic aura mechanism was introduced by Benford and Fahlén (1993) for the handling of connection establishment and removal, and enabler of the mechanisms that support focus and nimbus. In fact, connection establishment and removal can be seen as just special cases of the more general focus and nimbus mechanisms, so that an aura collision between an emitting object *A* and a receiving object *B* is deemed to take place exactly when *A* is within *B*'s focus and *B* is within *A*'s nimbus.

4.1 The Model

We shall explain the idea a little more precisely: With each object *A* is associated a point in a metric space *R* called *t*he r*oot space*. The point associated to *A* is its location, *loc(A)*. Objects come in two flavours, *emissive*, or *receptive*. To each object *A* is furthermore associated a set of subsets *S* of the root space, the *awareness levels* of *A*. Depending on whether the object concerned is emissive or receptive, an awareness level *S* corresponds to a nimbus or a focus level. We require of the awareness levels that they are totally ordered by set inclusion. That is, given any two awareness levels *S1* and *S2* of an object *A*, either *S1=S2*, or *S1* is a subset of *S2*, or *S2* is a subset of *S1*. This rules out branching of the awareness structure such as shown on fig. 5.i.

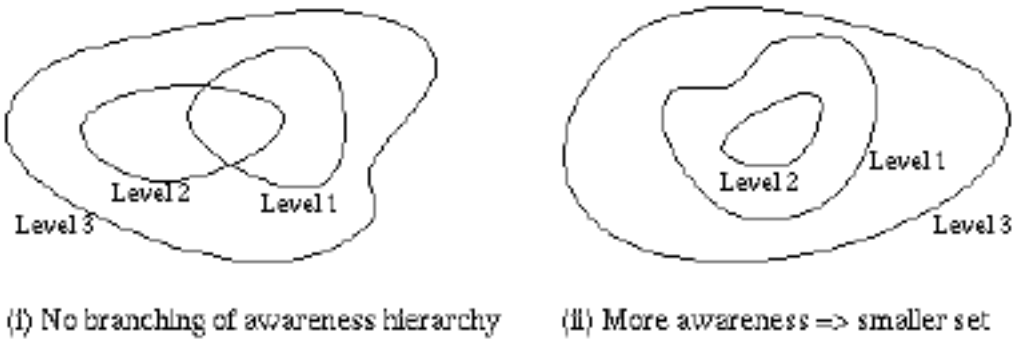
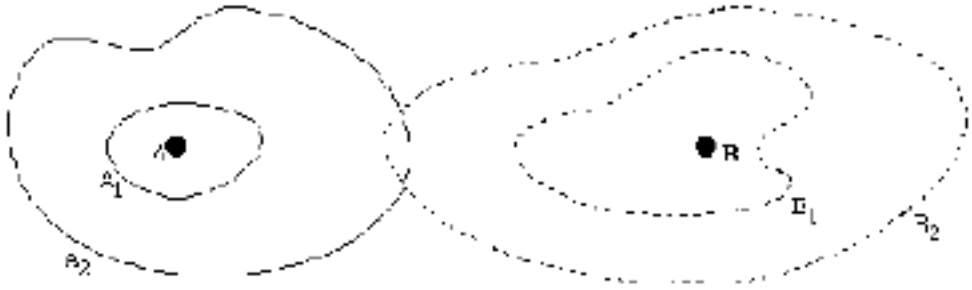


Figure 5: Awareness hierarchies

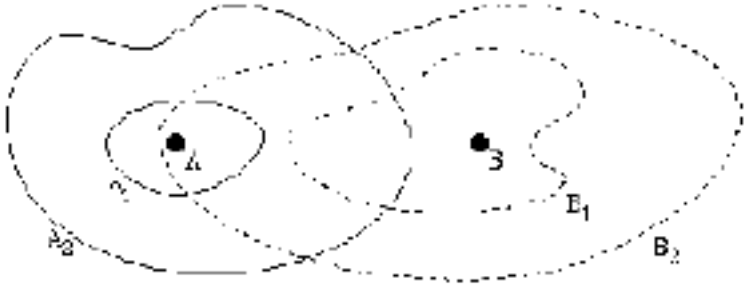
It is natural (though by no means required) to correlate with the set inclusion ordering on awareness levels an ordering *less-than* of ``increasing awareness'', such that *S1* is *less-than S2* (*S2* is a higher level of awareness than *S1*) if and only if *S2* is a subset of *S1* (*S2* is a smaller region of space than *S1*). Thus if larger index is taken to mean lower level of awareness also awareness hierarchies such as that shown on fig. 5.ii are ruled out. An equivalent way of stating these conditions is by saying that if *x* is a point in an awareness level *S1*, and *S2* is a

lower level of awareness than *S1* then *x* is also a point in *S2*, and if *x* is not a point in *S1*, and *S2* is a higher level of awareness than *S1* then neither is *x* a point in *S2*. Note that we do not make any assumptions as to the number (finite, countable, uncountable) or nature (open, closed, connected,...) of awareness levels.

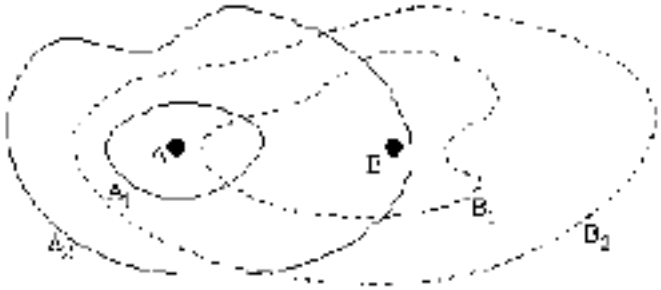
Using awareness levels we can now derive a natural correlate of aura collision where we deem an aura collision to take place between an emitting object *A* and a receptive object *B* just in case there is an awareness (nimbus) level *S(A)* of *A* such that *loc(B)* *S(A)* and an awareness (focus) level *S(B)* of *B* such that *loc(A)* *S(B)*. That is, intuitively, such that *B* is in some nimbus level of *A* and *A* in some focus level of *B*. If *S(A)* is the highest (i.e. smallest) nimbus level of *A* such that *loc(B)* *S(A)*, and *S(B)* is the highest focus level of *B* such that *loc(A)* *S(B)* then we talk of this state of affairs as an *S(A)*-*S(B)*-collision. An example is given on fig. 6.



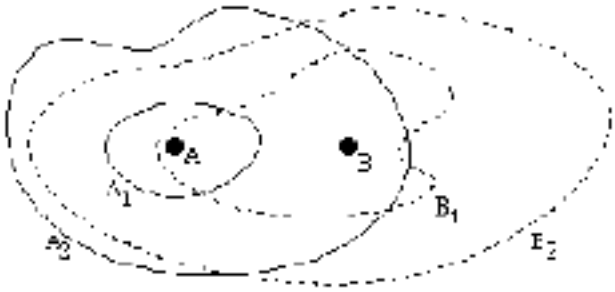
No collision



No collision: A in focus of B, B not in nimbus of A



Level 2 – level 2 collision



Level 2 – level 1 collision

Figure 6: The collision mechanism

Here *A* is assumed to be an emissive, and *B* a receptive object. Initially no collision is present since *A* is not in a focus of *B* and *B* not in a nimbus of *A*. Even when *A* actually enters some focus level of *B*, as long as *B* is not in a nimbus of *A* no collision is deemed to take place. When *B* also enters a nimbus of *A* an *A2-B2*-collision takes place, and when subsequently *A* enters *B* focus level *B1* a state of *A2-B1*-collision results. Summarising we can conceive of the present awareness model as an extension of the aura model where

1. auras are emissive or receptive,
2. aura collisions are determined with respect to the relative positions aura-object and object-aura, and
3. auras are allowed be nested.

So far nothing has been said about what effect, if any, awareness levels have on the communication that takes place between objects. We conceive of communication as a process of rippling through pieces of information from emitting object down through decreasing levels of nimbus until a state of collision is encountered whereupon the process is reverted on the receiving side. That is, information ripples up through increasing levels of focus until it is passed to the receiving object. Thus, to interpret awareness it is natural to associate with each awareness level *S* a filtering action *f(S)* that is applied to information passing through that level. Relating to the situations on fig. 6 filtering actions are indicated on fig. 7 by squiggly lines from inwards out in the case of nimbii (and outwards in, in the case of focii).

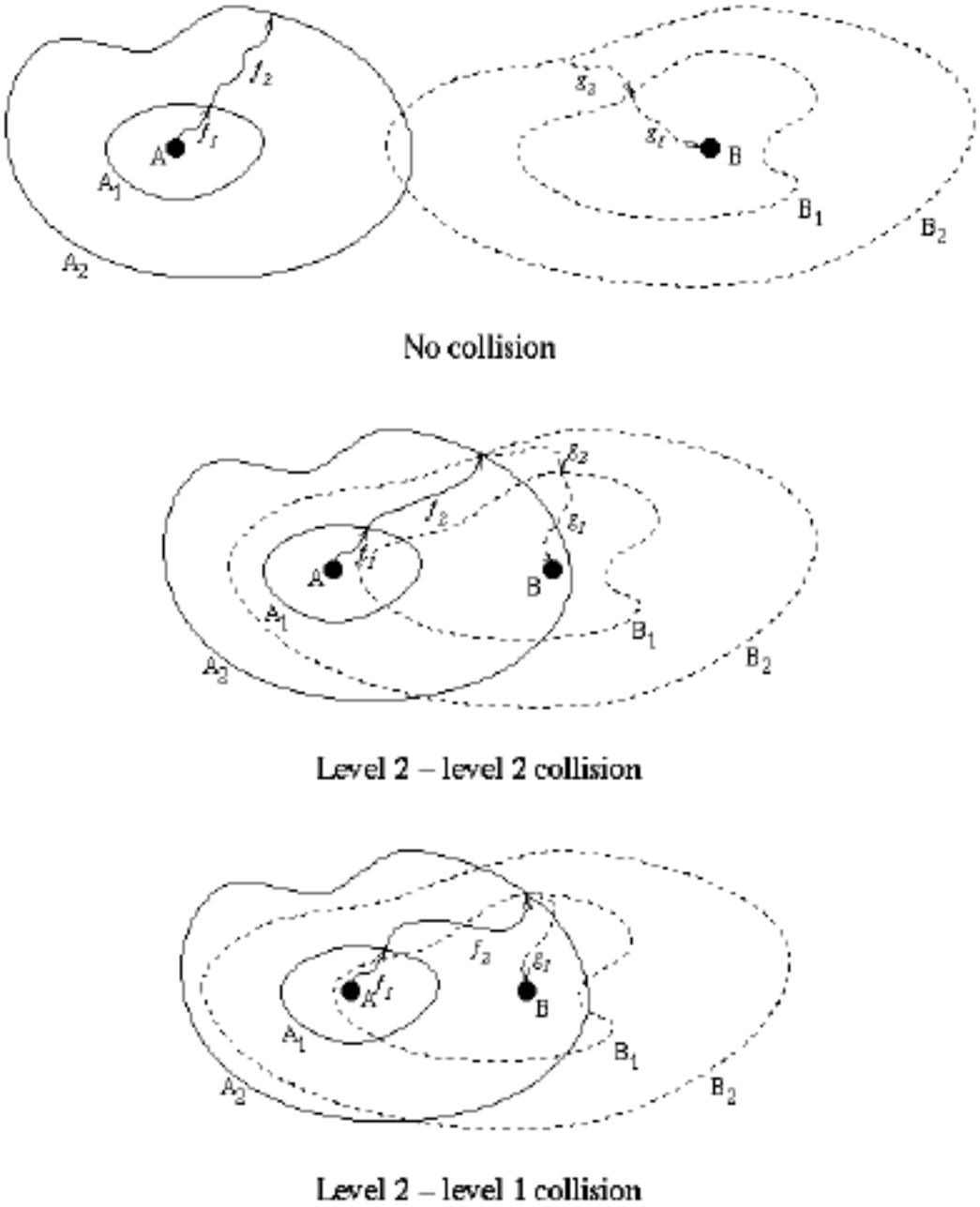


Figure 7: Filtering actions

In the absence of collision no information is passed between the nimbus-focus stacks involved. In the case of an *A2*-*B2*-collision the total filtering action is determined as the composite *g1* o *g2* o *f2* o *f1* of the individual filtering actions. That is, if *x* is a piece of information emitted from *A* then *g1(g2(f2(f1(x))))* is the corresponding piece of information received by *B*. In case of an *A2*-*B1*-collision the composite filtering action is then *g1* o *f2* o *f1*. Concretely, filtering actions can be of highly varying types. They can range from structure abstractions to attenuation of sound or blurring of video. It is worth noting that in general

filtering actions will *not* preserve the property of belonging to a particular medium. For instance, a filtering action on live video may produce stills, text, or iconic information.

4.2 The Specification

Proceeding to present the specification, for the sake of the presentation we restrict attention to the much simplified situation of just one emitting and one receiving object. There is no difficulty in principle in dealing with the more general problem of multiple emitters and receivers. This is discussed briefly below. The complete system is composed of one connection object together with collections of ``emitter'' and ``receiver'' objects. Each emitter/receiver corresponds to one awareness level. The emitters are all chained to form one awareness hierarchy. The similar situation applies to the receivers. The connection object connects exactly one (the ``active'') emitter with exactly one (``active'') receiver according to how the connection conceives the current state of collision to be. Fig. 8 depicts the way a connection object is linked to its immediate neighbours.

Figure 8: Connection object linkage structure

A connection object receives data input from its neighbouring emitter object through the local channel `in` and passes it on to the receiver object through the channel `out` . Position of emitter and receiver objects are monitored through the local channels `in-pos-e` (for the emitter) and `in-pos-r` (for the receiver). These channels are all those owned by a connection object, and we can therefore describe its type as follows:

```
type ‘a ‘b CONNECTION =
     channels
          in?’a,
          out!’a,
          in-pos-e?’b,
          in-pos-r?’b
      end;
```

We make no specific assumptions concerning the nature of data or position information, and we therefore represent those types by the type variables `‘a` and `‘b`. In addition to the channels owned by connection objects each emitter/receiver

owns three channels for collision checking: a channel `check-coll` along which a position is passed in order to check whether or not that position is in a state of collision with the current emitter/receiver, and channels `colliding` and `not-colliding` along which references to emitters/receivers are passed. These channels are used by both connection objects as well as by other emitters/receivers.

Fig. 9 shows the way an active emitter object is linked with its neighbours.

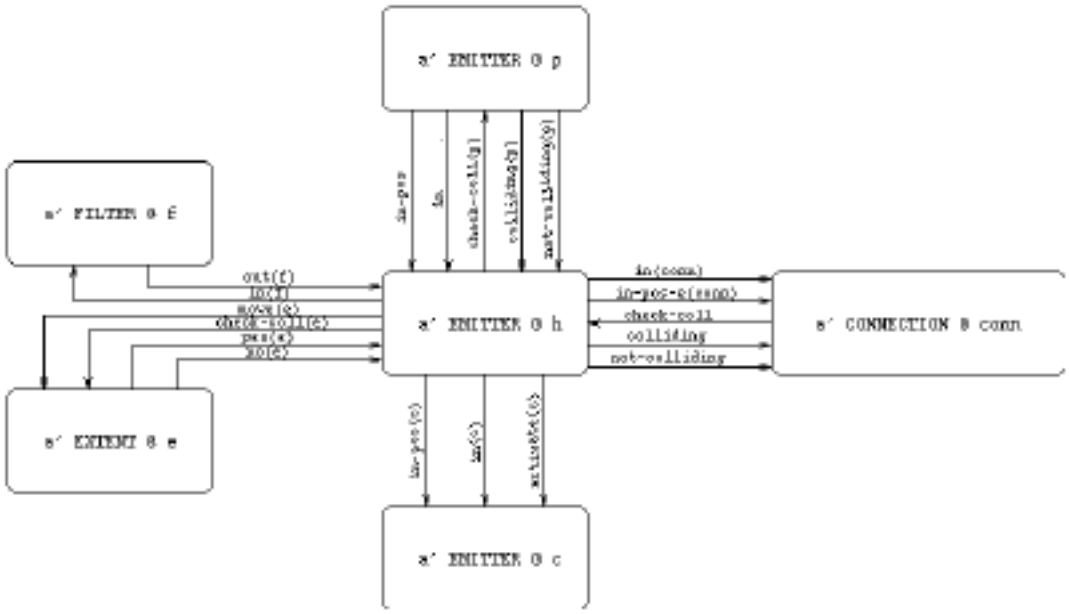


Figure 9: Emitter object linkage structure

We have already described the links connecting an active emitter with the connection. Each emitter, active or not, has attached to it a ``filter'' object responsible for the filtering action associated with the current awareness level, and an ``extent'' object that can be queried whether a position is within the extent of the current awareness level. The types of filter and extent objects are as follows:

```
type 'a FILTER =
     channels
          in?'a,
          out!'a
      end;

type 'b EXTENT =
       channels
          move?'b,
          check-coll!'b.(yes! + no!)
```

```
      end;
```

Emitters are chained to form an awareness hierarchy. Thus each emitter is connected to its parent (of higher awareness level) through which it receives its input and position information, and which it may need to query to see if the state of collision needs to be changed. Querying for collision entails passing position information plus a reference to the connection. Emitters are also connected downwards to their children to which they may pass data and position input. Furthermore an active emitter may, if it finds itself no longer in a state of collision, need to activate its child, passing to it a connection reference. Thus we arrive at the following type for emitters:

```
type ‘a ‘b EMITTER =
       channels
       in?’a,
          in-pos?’b,
          activate?’a ‘b CONNECTION,
          check-coll?’b x ‘a ‘b CONNECTION.
                  (colliding!’a ‘b EMITTER +
                   not-colliding!’a ‘b EMITTER)
 end;
```

Similarly, the type of receivers is the following:

```
type ‘a ‘b RECEIVER =
       channels
          out!’a,
          in-pos?’b,
          activate?’a ‘b CONNECTION,
          check-coll?’b x ‘a ‘b CONNECTION.
              (colliding!’a ‘b RECEIVER +
               not-colliding!’a ‘b RECEIVER)
      end;
```

Now for the dynamics we first consider connection objects. These are very simple. They maintain references to their neighbours as well as their current positions. Data input is passed from emitter to receiver, position changes are recorded, or else a connection can choose to query its neighbouring emitter or receiver for changes in collision status, receiving back emitter/receiver references to replace the earlier ones.

```
object
      Connection(‘a ‘b EMITTER @ e,
                 ‘b @ p-e,
                 ‘a ‘b RECEIVER @ r,
                 ‘b @ p-r): ‘a ‘b CONNECTION @ c =
           in?x.out!x.Conection(e,p-e,r,p-r) @ c
               +
 in-pos-e?new-p-e.
 Connection(e,new-p-e,r,p-r) @ c
               +
 in-pos-r?new-p-r.
 Connection(e,p-e,r,new-p-r) @ c
               +
           check-coll(r)!(p-e,c).
 (colliding(r)?new-r.
 Connection(e,p-e,new-r,p-r) @ c
     +
 not-colliding(r)?new-r.
 Connection(e,p-e,new-r,p-r) @ c)
               +
 check-coll(e)!(p-r,c).
 (colliding(e)?new-e.
 Connection(new-e,p-e,r,p-r) @ c
               +
       not-colliding(e)?new-e.
 Connection(new-e,p-e,r,p-r) @ c)
```

Next for emitters. A general emitter has both a parent and a child emitter. The cases of ``top’’ emitters lacking parents, and ``bottom’’ emitters lacking children are easy special cases left to the reader. General emitters come in two versions: an active and a passive one, the difference being where incoming information is passed and how collision checking is handled. We here describe only the situation for active emitters. Data is received from the parent and passed to the connection. Position information is passed on to both connection and descendant. For collision checking the position incoming from the connection is passed on the extent object. If a state of collision continues to hold the parent emitter is queried for collision. If the parent also collides with the object the connection is informed of the parent reference and the emitter turns passive. If, on the other hand, the parent is not colliding no change ensues. Finally, in case the state of collision with the current emitter has ceased to hold, the descendant emitter is activated, the

connection is informed of the identity of the new active emitter, and the current emitter turns passive. Or more concisely:

```
object
 ActiveEmitter(‘a ‘b EMITTER @ p,
               ‘a ‘b EMITTER @ c,
               EXTENT @ e,
 ‘a FILTER @ f,
 ‘a ‘b CONNECTION @ conn):
 ‘a ‘b EMITTER @ h =
 in?x.(f)!x.out(f)?y.in(conn)!y.
 ActiveEmitter(p,c,e,f,conn) @ h
 +
 in-pos?pos.move(e)!pos.in-pos-e(conn)!pos.
 in-pos(c)!pos.ActiveEmitter(p,c,e,f,conn) @ h
     +
 check-coll?(pos,conn1).check-coll(e)!pos.
 (yes(e)?.check-coll(p)!(pos,conn).
       (colliding(p)?new-e.colliding!new-e.
       PassiveEmitter(p,c,e,f) @ h
           +
 not-colliding(p)?e1.colliding!h.
 ActiveEmitter(p,c,e,f,conn) @ h)
       +
 no(e)?.activate(c)!conn.not-colliding!c.
 PassiveEmitter(p,c,e,f) @ h)
and
 PassiveEmitter(‘a ‘b EMITTER @ p,
               ‘a ‘b EMITTER @ c,
               EXTENT @ e,
               ‘a FILTER @ f):
               ‘a ‘b EMITTER @ h =
    in?x.in(f)!x.out(f)?y.in(c)!y.
 PassiveEmitter(p,c,e,f) @ h
 +
 in-pos?pos.move(e)!pos.inpos(c)!pos.
 PassiveEmitter(p,c,e,f) @ h
 +
    activate?conn.ActiveEmitter(p,c,e,f,conn) @ h
```

```
      +
    check-coll?(pos,conn).check-coll(e)!pos.
  (yes(e)?.colliding!h.
  ActiveEmitter(p,c,e,f,conn) @ h
       +
  no(e)?.not-colliding!c.
  PassiveEmitter(p,c,e,f) @ h)
```

For receivers the situation is somewhat more complex since they need to pass information both downwards (position information) and upwards (data output). Deadlocks may result as a consequence. For instance consider the situation of the connection having accepted data input from the emitting side and waiting to pass this on to the receiving side, and the receiving side having accepted a position update waiting to pass this downwards and on to the connection. An easy solution is to give one of the directions priority over the other. Here we choose to give the upwards direction priority over the downwards one. Unfortunately this results in a somewhat more cumbersome code as shown below. Whether this type of situation is common enough to warrant a more general account, and how such an account should look is an issue for further investigation.

```
object
 ActiveReceiver(‘a ‘b RECEIVER @ p,
                ‘a ‘b RECEIVER @ c,
                EXTENT @ e,
                ‘a FILTER @ f,
 ‘a ‘b CONNECTION @ conn):
 ‘a ‘b RECEIVER @ h =
 out(conn)?x.in(f)!x.out(f)?y.
 ActiveReceiver(p,c,e,f,conn) @ h
     +
 in-pos?pos.move(e)!pos.
 TellConn(pos,p,c,e,f,conn) @ h
 +
 check-coll?(pos,conn1).check-coll(e)!pos.
     (yes(e)?.check-coll(p)!(pos,conn).
       (colliding(p)?new-r.colliding!new-r.
                       PassiveReceiver(p,c,e,f) @ h
 +
 not-colliding(p)?r1.colliding!h.
                   ActiveReceiver(p,c,e,f,conn) @ h)
```

```
 +
 no(e)?.activate(c)!conn.not-colliding!c.
                       PassiveReceiver(p,c,e,f) @ h)
and
 TellConn(‘b @ pos,
          ‘a ‘b RECEIVER @ p,
          ‘a ‘b RECEIVER @ c,
          EXTENT @ e,
          ‘a FILTER @ f,
 ‘a ‘b CONNECTION @ conn):
 ‘a ‘b RECEIVER @ h =
 out(conn)?x.in(f)!x.out(f)?y.out!y.
 TellConn(pos,p,c,e,f,conn) @ h
 +
    in-pos-r(conn)!pos.TellChild(pos,p,c,e,f,conn) @ h
and
 TellChild(‘b @ pos,
          ‘a ‘b RECEIVER @ p,
          ‘a ‘b RECEIVER @ c,
          EXTENT @ e,
          ‘a FILTER @ f,
 ‘a ‘b CONNECTION @ conn):
 ‘a ‘b RECEIVER @ h =
 out(conn)?x.in(f)!x.out(f)?y.out!y.
 TellChild(pos,p,c,e,f,conn) @ h
 +
 in-pos(c)!pos.ActiveReceiver(p,c,e,f,conn) @ h
and
 PassiveReceiver(‘a ‘b RECEIVER @ p,
                 ‘a ‘b RECEIVER @ c,
                 EXTENT @ e,
                 ‘a FILTER @ f):
 ‘a ‘b RECEIVER @ h =
 out(c)?x.in(f)!x.out(f)?y.out!y.
 PassiveReceiver(p,c,e,f) @ h
 +
 in-pos?pos.move(e)!pos.PTellChild(pos,p,c,e,f) @ h
     +
    activate?conn.ActiveReceiver(p,c,e,f,conn) @ h
```

```
and
 PTellChild(‘b @ pos,
           ‘a ‘b RECEIVER @ p,
           ‘a ‘b RECEIVER @ c,
       EXTENT @ e,
           ‘a FILTER @ f):
           ‘a ‘b RECEIVER @ h =
 out(c)?x.in(f)!x.out(f)?y.out!y.
 PTellChild(pos,p,c,e,f) @ h
 +
 in-pos(c)!pos.PassiveReceiver(p,c,e,f) @ h
```

This concludes the example awareness model specification. It is very rudimentary indeed, and should be extended and modified in various ways. The structure of the specification could be improved by using product types to hide the collision checking mechanisms from the outside world. Concerning extensions of most immediate concern is to consider multiple emitters and receivers. To accommodate this some form of multicasting needs to be added. Further afield mechanisms for dynamically modifying, combining, creating and removing awareness levels should be considered. Note that there is no significant loss in generality in considering just emissive or receptive auras, or restricting attention to a single medium. A primitive mechanism is needed to glue together awareness hierarchies belonging to different media, as well as receptive and emissive awareness hierarchies. Such a gluing mechanism could furthermore be used to give the effect of adapters (c. f. Benford and Fahlén, 1993) in analogy with e.g. loudspeakers, or binoculars.

On a somewhat deeper level it may be of considerable value to divorce the concept of space governing awareness structures from its representation as e.g. 3D space in a virtual reality environment such as DIVE (Carlsson and Hagsand, 1992). In this manner - by relying on a more abstract notion of ``position'' - a potentially much finer control of awareness level is possible than that achievable by just using ``position in 3D space''. Note that in this manner 3D representation position information may become just one medium along with whatever others that may be considered. An example application concerns the notion of boundary that can conceivably be managed already in the current rudimentary model quite simply by mapping 3D representation space onto a suitable non-Euclidean space. As another example, by relying on a more abstract underlying notion of space it may be possible to let the current rudimentary model capture awareness structures that can depend on other parameters of the representation space than just direction and medium, say position, or movement.

5. Conclusion and Future Work

We have argued that a critical dimension in the handling of change in computer based systems for cooperative work is whether facilities for change should be (explicitly) embedded into systems, or whether change should be handled in a uniform and global manner, for instance by a process of editing and recompiling programs or scripts on the fly. While more general the latter approach is unsatisfactory in many situations since it affords too little in the way of control and structure. On the other hand, embedding facilities for change into computer systems is a difficult task since first of all the concept of change is a highly unruly and poorly understood one, and in each instance it needs to be considered, say, who is empowered to perform what changes, what changes are to be handled, and in what ways the statics and dynamics of the system are affected by the change.

Having made this point the main contribution of the present paper is to propose a formal basis for the description of dynamically modifiable concurrent objects, and to explore its applicability in the field of CSCW. The types of change that objects can be exposed to are superficially rather modest: They can be dynamically created and their interconnection structure can be dynamically altered. We considered two examples to illustrate the basic ideas: An adaptable conversation manager, and a rudimentary awareness model along the lines of Benford and Fahlén (1993).

We do not regard the notation as presented here as very final. More work, both theoretical and practical, is needed to identify the `right'' primitives and their relation to more well-known and standard concepts from object-oriented languages. For instance an analysis of inheritance in terms of the ideas presented here should be attempted. Also left for future work is the development of analysis tools based on the notation. The specification language is based on the pi-calculus of Milner, Parrow, and Walker (1992). Other object-based extensions of the pi-calculus have been considered by Walker (1994) and Jones (1993). Algorithms exists for deciding properties such as deadlock freedom, equivalence, safety, or liveness properties for large classes of processes (Dam, 1993; Dam, 1994), and rudimentary tools for automatically performing analyses are beginning to emerge (Victor and Moller, 1994). The class of processes that can be captured is, however, still too small. It includes processes that are, in essence, finite state up to naming of communication channels. This includes the 1 emitter - 1 receiver awareness model presented above but not the adaptable conversation manager spawning, as it can, an unbounded number of transition objects. In future work we intend to expose examples along the lines of those presented above to analysis to investigate the difficulties involved in deriving interesting properties using current tools, as well as to suggest practically useful extensions.

Acknowledgments

Thanks to Carl Brown and Lars-Åke Fredlund for discussion and comments.

References

Benford, Steve and Fahlen, Lennart: “A spatial model of interaction in large virtual environments,” in *Proceedings of ECSCW’93* , Milan, September 1993.

Carlsson , Christer and Hagsand, Olof: “The MultiG distributed interactive virtual environment,” in *Proceedings of the 5th MultiG Workshop* , Stockholm, December 1992.

Dam, Mads: “Model checking mobile processes,” in *Proc. CONCUR’93, Lecture Notes in Computer Science* (715) 22-36, 1993. Full version in SICS report RR94:1, 1994.

Dam, Mads: “On the decidability of process equivalences for the -calculus,” submitted for publication , 1994.

Flores, Fernando, Michael Graves, Brad Hartfield, and Terry Winograd: “Computer Systems and the Design of Organizational Interaction,” *ACM Transactions on Office Information Systems*, vol. 6, no. 2, April 1988, pp. 153-172.

Heath, Christian, and Paul Luff: “Collaboration and Control. Crisis Management and Multimedia Technology in London Underground Control Rooms,” *Computer Supported Cooperative Work (CSCW). An International Journal*, vol. 1, no. 1-2, 1992, pp. 69-94.

Johnson, Philip: “Supporting Exploratory CSCW with the EGRET Framework,” in J. Turner and *Cooperative Work, Toronto, Canada, October 31 to November 4, 1992*, ACM Press, New York, 1992, pp. 298-305.

Jones, Cliff B.: “A pi-calculus semantics for an object-based design notation,” in *Proceedings CONCUR’93, Lecture Notes in Computer Science* (715) 158-172, 1993.

Kaplan, Simon M., William J. Tolone, Douglas P. Bogia, and Celsina Bignoli: “Flexible, Active Support for Collaborative Work with Conversation Builder,” in J. Turner and R. Kraut (eds.): *CSCW ‘92. Proceedings of the Conference on Computer-Supported Cooperative Work, Toronto, Canada, October 31 to November 4, 1992*, ACM Press, New York, 1992, pp. 378-385.

Malone, Thomas W., Kum-Yew Lai, and Christopher Fry: “Experiments with Oval: A Radically Tailorable Tool for Cooperative Work,” in J. Turner and R. Kraut (eds.): *CSCW ‘92. Proceedings of the Conference on Computer-Supported Cooperative Work, Toronto, Canada, October 31 to November 4, 1992*, ACM Press, New York, 1992, pp. 289-297.

Milner, Robin: “The polyadic pi-calculus: A tutorial,” Technical Report ECS-LFCS-91-180, Laboratory for the Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1991.

Milner, Robin, Parrow, Joachim and Walker, David: “A calculus of mobile processes, I and II ,” *Information and Computation* ,(100) 1-40 and 41-77, 1992.

Orava, Fredrik and Parrow, Joachim: “An algebraic verification of a mobile network,” *Formal Aspects of Computing* (4) 497--543, 1992.

Paulson, Larry: “*ML for the Working Programmer*,” Cambridge University Press, 1991.

Sangiorgi, Davide: “From pi-calculus to higher-order pi-calculus - and back,” in *Proc. TAPSOFT’93*, 1993.

Schmidt, Kjeld: “Modes and Mechanisms of Interaction in Cooperative Work,” in C. Simone and K. Schmidt (eds.): *Computational Mechanisms of Interaction for CSCW*, Computing Department, Lancaster University, Lancaster, U.K., 1993, pp. 21-104. - [COMIC Deliverable 3.1. Available via anonymous FTP from ftp.comp.lancs.ac.uk].

Schmidt, Kjeld, Carla Simone, Peter Carstensen, Betty Hewitt, and Carsten Sørensen: “Computational Mechanisms of Interaction: Notations and Facilities,” in C. Simone and K. Schmidt (eds.): *Computational Mechanisms of Interaction for CSCW*, Computing Department, Lancaster University, Lancaster, U.K., 1993, pp. 109-164. - [COMIC Deliverable 3.1. Available via anonymous FTP from ftp.comp.lancs.ac.uk].

Simone, Carla, Pozzoli, A, Schmidt, Kjeld and Hewitt, Betty: “An architecture for malleable and linkable mechanisms of interaction,” manuscript , 1994.

Victor, Björn and Moller, Faron: “The mobility workbench: A tool for the pi-calculus,” in *Proc. Conf. Computer Aided Verification* , 1994.

Walker, David: “Objects in the pi-calculus,” *Information and Computation* , 1994 (To appear).

Young , R. E. (ed.): “COSMOS . Specification of a configurable structured message passing system,” Queen Mary College, London , 1989.

Architectures and notations for computational mechanisms of interaction

Prototyping the C4SO Architecture

Leandro Navarro

UPC

The Shared Object Services and the Shared Interface Services (SOS/SIS: S*S) is an architecture to computer support the complex interactions of multiple actors working in a complex work arrangement. The S*S has grown as a result of the work in the COMIC Strand 4 Sharing Objects group (C4SO) from a set of prototype systems to be an architectural model of the components, primitives, dependencies, flows of information among the components of such large scale computer system. In other words, it provides a notation(s) at the semantic level of cooperative work to design and incorporate different mechanisms of interaction built on a common set of primitives, a common notation in order to avoid artificial boundaries and support articulation work [Milano-3-3].

An overview of the architecture with a focus on the Resource Manager, a component that provides primitives to manipulate the objects of articulation work and reduce the complexity of articulation work by providing orderly access to resources in their organisational context.

The Aleph prototype system and the Aleph-Tcl are a prototype system and notation with the required malleability and linkability to specify mechanisms of interaction based on the C4SO architecture. It provides a possible platform where to implement the concepts developed in COMIC Strand 3.

1. Introduction

The *Shared Object Services* and the *Shared Interface Services* (SOS/SIS: S*S) is an architecture to computer support the complex interactions of multiple actors working in a complex work arrangement. The S*S has grown as a result of the work in the COMIC Strand 4 Sharing Objects group (C4SO) from a set of prototype systems to be an architectural model of the components, primitives, dependencies, flows of information among the components of such large scale computer system. In other words, it provides a notation(s) to design and incorporate different mechanisms of interaction built on a common set of primitives, a common notation in order to avoid artificial boundaries and support articulation work [Milano-3-3] [Malone 92]. This paper tries to explore the contact between the work on a malleable and linkable architecture in COMIC Strand 3 and the S*S architecture, their notation and specially the work on the Aleph prototype at UPC.

The S*S notation incorporates primitives at various semantic levels, some of which are applicable to narrow application domains, while others are applicable to the wider domain of cooperative work. This is the rationale behind the taxonomy of S*S primitives according to layers, which can be easily extended to include additional primitives in the form of new components (*applications* or *managers*, depending on the layer) which will provide new primitives and, at the same time, will use existing ones.

The goal of the S*S architecture is to enable the construction of computer systems to support large scale, complex cooperative work environments. Even though there are successful CSCW applications addressing narrow domains with low degree of complexity, complex work environments may not be supported in practice by a number of independent CSCW applications.

Up to a certain threshold of complexity, work articulation can be addressed by human actors by means of the modes of interaction of everyday social life [Deliv 3.1]. However, the complex work environments most organisations provide is beyond human scale. Complexity comes from several facts: (1) *actors are semi-autonomous* in terms of strategies, heuristics, conceptualisations, goals, motives, etc. and (2) *distributed character of cooperative work*: distribution of activities in time and space, the number of participants in the cooperative ensemble, structural complexity posed by the field of work [Risø-3-10].

Therefore, in complex work environments the task of articulating the complexly interdependent and distributed activities is beyond human social and communication skills. The S*S provides mechanisms, in the form of Managers, that stipulate and mediate both, work and articulation work, and thereby reducing the complexity of developing new tools to support cooperative work, and reducing the complexity of articulation work.

The S*S is a framework of primitives at the semantic level of cooperative work (both work and articulation work) to *develop* and *apply* domain-specific applications as well as new mechanisms:

- Component design and development is deliberately oriented to be expressed in terms of the primitives provided by other components so that fluid interrelationships of cooperative work are reflected in fluid interrelationships among S*S components. Therefore new mechanisms or applications are described in a notation based on the primitives provided by the rest of components.

  For example, resources can be reached by means of the primitives provided by the Resource Manager, which is a component that provides access to resources taking into account the entities involved with or related to resources such as actors, other tasks, the organisational context, etc. Events produced by a component are handed on the Event Manager to be transformed or distributed to other interested components. Presentation of shared objects is expressed in terms of the primitives of the presentation service, the SIS, that will present objects taking into account the cooperative

issues in the user interface: support for joint usage of applications, and support for awareness concerning object sharing.

- An S*S environment provides a general notation to express to people the mutual dependencies of a large number of actors and activities through the user interface, that is the SIS; and to support, enable, grant, etc. the interrelation of multiple applications in a particular setting without posing any barrier on the articulation work. Articulation is stipulated and mediated by the SOS components. These components can be manipulated independently of the state of the field of work, i.e. symbolic artifacts, and they provide affordances to and impose constrains on cooperative work. *They are abstract devices at different semantic levels*: at the level of sharing objects, at the level of domain-specific work, and at the level of articulation work. The last group of computational devices are also referred to as *computational mechanisms of interaction*.

  For example, in the UPC Prototype system, actors may activate the Finder control panel, an application that provides affordances to respecify the behaviour of the Finder component in a cooperative manner (It is a S*S application, presented by the SIS). Functional primitives of this Control Panel enable to manipulate or browse through the services, potential partners, resources visible to an actor, for a given task, etc. indexed by contexts. Primitives of this panel are navigate, reserve, move, name, consume, characterise, relate, activate, join, leave, enroll, define category, classify, etc.

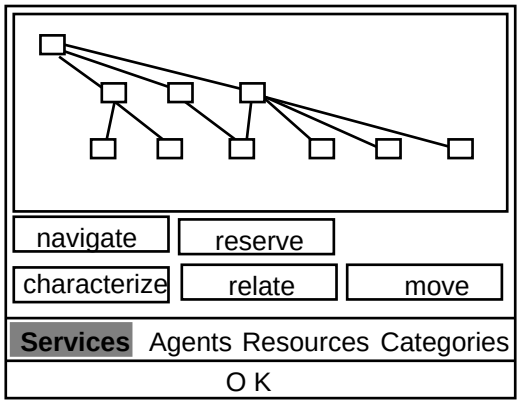


Fig. 1. These primitives are at the semantic level of articulation work. The same set of primitives are presented to any component in the S*S by an internal invocation mechanism. In this window, the behaviour of the Finder can be manipulated independently of the work going on[1].

This way, domain-specific applications can be easily specified and designed in terms of the primitives provided by other S*S components (or mechanisms). A common notation at the same semantic level, coupled with the fact that a large number of primitives are provided by common components, provides flexibility

1 The Finder Control Panel is currently under development as part of Aleph, the UPC Prototype System.

and interoperability across applications. This facilitates the integration of multiple applications and therefore it allows the articulation of cooperative work with respect to these applications.

The S*S architecture of components provides symbolic artifacts at the semantic level of sharing, domain-specific work, and articulation work. They are linked by the SOS architecture specification, and presented to people in terms of the SIS. In addition, there is a textual computational notation under development (Aleph-Tcl) to support the specification and execution of mechanisms of interaction and domain-specific cooperative applications.

The rest of the document is structured as follows:

Section 2: “The S*S Architecture”. A brief introduction of the architecture being developed in the COMIC Strand 4 Shared Object and Interface Service (C4SO)

Section 3: “The Resource Manager”. This is the key component of the Aleph system. The Resource Manager reduces the complexity of articulation work by stipulating and mediating the articulation of cooperative tasks by providing orderly access to resources in an organisational context.

Section 4: “The Aleph-Tcl Language”. This is an embedded interpreted language that provides ways to combine not only the primitives provided by the Resource Manager, but also the primitives provided by other S*S components. This enables to build malleable Mechanisms of Interaction.

Section 5: “Malleability”. Changes to mechanisms can be effected in a cooperative manner by the control panel mechanism. The extent and applicability of changes is given the interpreted nature of the language and the mechanisms of propagation of events on the S*S environment.

Section 6: “Linkability”. The S*S architecture and notations such as Aleph-Tcl provide easy support for linking components and mechanisms, and delegate to the S*S components the burden of articulation work.

Section 7: “Conclusions”. Basically, the S*S provides an flexible architecture with primitives at the level of cooperative work, an cooperative interface notation (SIS) and a malleable textual notation (Aleph-Tcl) to build flexible and linkable mechanisms of interaction.

2. The S*S Architecture

The Shared Object and Interface Service (SOS and SIS) is an architecture intended to support multiple actors using a multiplicity of applications to work cooperatively in a complex work arrangement. It is an evolving set of components and primitives orchestrated to rely on each other, i.e. their environment, and therefore facilitate the design of new components (end user Application or Managers) based on the primitives provided by existing ones.

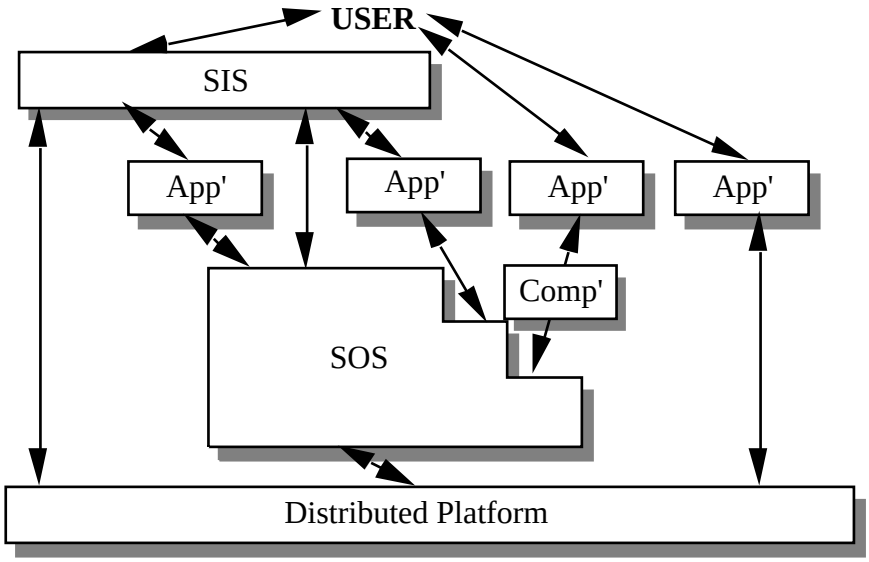


Fig 2. The SIS and the SOS provides applications, or computational components in general, an environment of primitives or functions at the semantic level of cooperative work.

The Shared Interface Service (SIS) takes care of some cooperative issues at the User Interface level such as support for joint usage of applications, and support for awareness concerning object sharing.

The Shared Object Service (SOS) architecture may be decomposed into three distinguished layers. Components on each layer provide primitives to support distributed cooperation, shared awareness and organisational support. Applications using the SOS may access the services of whatever layer is appropriate and suited to serve their specific needs.

The lowest layer consists of the basic mechanisms built on top of the computational infrastructure (distribution platform, operating system, etc.). These components provide primitives to build upper layer primitives. A second layer of core services provides general mechanisms to support cooperative applications. An extensible set of common services forms the upper layer and implements refined sets of primitives, domain specific common primitives, optional primitives, etc., i.e. those components that provide primitives which are not essential for any cooperative task, but that may be shared between multiple applications, and that are key for integration (e.g. a common history service enables sharing history logs across a range of applications).

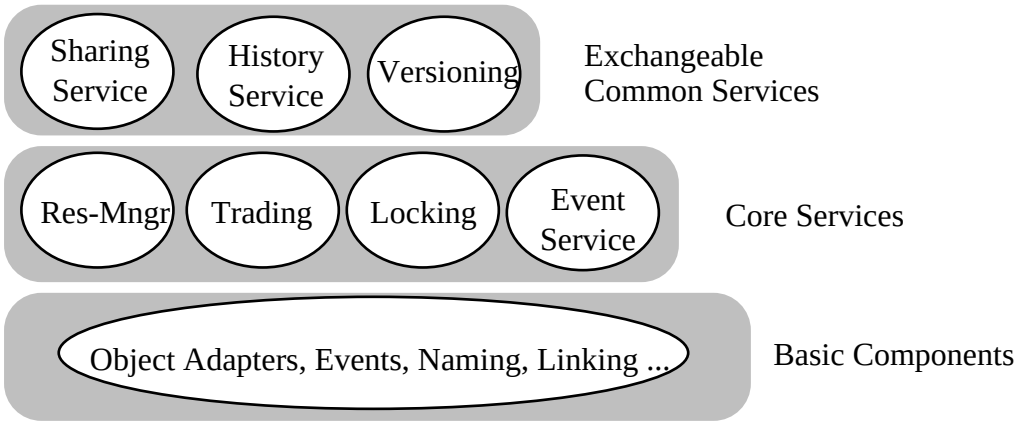


Fig. 3 The SOS layered architecture with the current set of services

This set of components and primitives are not intended to be exhaustive. It reflects the issues addressed by the current prototype systems and it is intended to be revised as a result of the work in the COMIC project. The current set of functions are: locking and versioning (reaching an agreement on sharing and modifying objects), history (a compilation of past actions), resource management (organisational restrictions, management of dynamic and scarce resources), trading (a mechanism to support articulation), event circulation (to propagate information about relevant actions and changes), awareness (knowledge about each other's actions). [UPC-4-2]

This model allows the application designer to access all different components at any level and select which primitives are going to be combined in a notation to specify and design a new application.

3. The Resource Manager

Resource Management is a function required to orderly access to resources for cooperative work, and any other entity in a organised way. This means to know and apply the knowledge about the organisation and the relationships among entities where work is situated.

The *Resource Manager* is an entity who mediates and supports articulation work situated in a Organisational Context. It applies and observes organisational policies to establish bindings among entities in an environment where entities are scarce, there is competition to use them, and the environment is large and changing.

The Resource Manager is the referee of conflicting requests coming from different agents that must be resolved in terms of the organisational policies. It has to know about any entities in their environment and how those entities are structured. It is also responsible for keeping track of the dynamics of the environment and it has to be able to adapt to it. Therefore, it reduces the complexity of articulation work by stipulating access to resources, i.e. reducing

local control that actors have over articulation, or supporting articulation work with incomplete information.

Two functions are specially important in Resource Management: Trading and Binding.

Trading is a function to resolve at any given time the most adequate server for every request. In a large scale organisation, entities change the way they work, the service they provide, their location, new entities appear. Trading is the function to find resources that best fit to our needs, and isolate from changes of name, location, and even from destruction and creation of new resources.

On one side, users provide a descriptive name of the entity they need, on the other side, there are entities that may be contacted to. The interaction between both sides is mediated by the *Finder*.

Binding is a function required to establish relationships among entities in a proper way. Binding two different entities frequently requires to insert cushion objects (object adapters or interceptors) to adapt, expand, transform, monitor, supervise, coordinate the interactions between entities.

The *binder* is the expert on contact making. It decides how entity relationships have to be configured to comply with the requirements of the parties involved and the organisational requirements for supervision.

The primitives and the notation for Resource Management are based on a modelling approach presented in [Deliv 1.1] which stipulates the objects of articulation work: agents, roles, actions, responsibilities, conceptual structures, resources. Section 4 discusses on Resource Management primitives and a notation for them that can be easily extended to the whole S*S.

4. The Aleph-Tcl Language

The Resource Manager has to provide primitives to provide agents access to resources, but also it has to provide primitives to configure and respecify their behaviour, and therefore the behaviour of the applications that rely on it. The Resource Manager is a Mechanism of Interaction so it provides manipulation primitives independently of the state of the field of work.

These primitives are presented in three ways: (1) to people in the form of a Control Panel, an application that provides affordances to monitor, configure and respecify the behaviour of the Resource Manager, or any other component, in a cooperative manner (It is a SOS application, presented by the SIS). It offers primitives at the appropriate semantic level to manipulate the Resource Manager independently of the work going on. (2) to any other S*S component, as a similar set of primitives that can be invoked. (3) as a language to specify, build or link the salient dimensions of resource management, or in general the salient dimensions of cooperative work. This is the Aleph-Tcl notation.

Changes in the Resource Manager originate not only from the respecification operations coming from the RM Control Panel, but also from the constant changes

of the organisational environment. Changes circulate in form of events that are received, processed and reflected into changes of the information, structures, policies, restrictions, etc. of the Resource Manager. These bindings (event-change) can also be specified in Aleph-Tcl which can easily change their behaviour while running since it is an interpreted language: changes are propagated by events and they are incorporated immediately by the Aleph-Tcl interpreter. This way, as soon as new request come to the resource manager, they are considered in terms of the recent changes.

This is how new applications, components or mechanisms can be easily specified and designed in terms of the primitives offered by the S*S environment as a whole. The S*S architecture provides symbolic artifacts with primitives at the semantic level of sharing, domain-specific work, and articulation work. Aleph-Tcl is a computational notation to specify and execute computational mechanisms.

Aleph-Tcl is based on the Tcl language. Tcl stands for “tool command language”. It is a simple scripting language for controlling and extending applications. It provides generic programming facilities that are useful for a variety of applications, such as variables, loops and procedures. Furthermore, Tcl is embeddable: its interpreter is implemented as a library of C procedures that can easily be incorporated into applications, and each application can extend the core TCL features with additional commands specific to that application [Ousterhout 93].

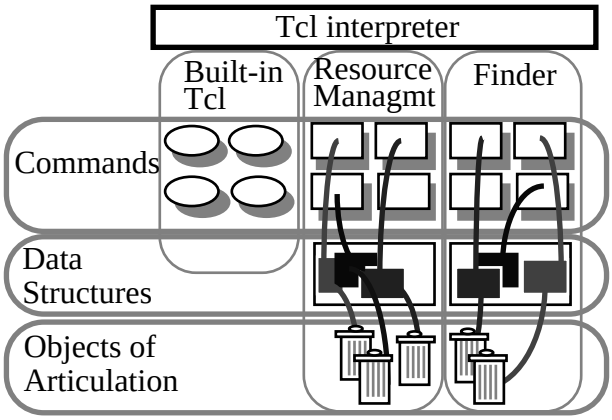


Fig. 5 Tcl is an embeddable and interpreted language which provides structures and basic commands. Aleph-Tcl extends the language with primitives at the level or cooperative work. Here it is shown that the Aleph-Tcl notation can be extended with primitives related to Resource Management and the Finder, but it can also incorporate primitives contributed by other components.

The RM language is an extension of the Tcl core primitives with Resource Management primitives. The result is a powerful scripting language at the required semantic level. Since it is a simple interpreted language, some modifications may be introduced while a script is running (commands sent to a script are executed by the interpreter). In the same way as the RM language is an extension of Tcl, Aleph-Tcl is an open language that can easily incorporate

primitives from diverse S*S components to support the construction of new components, mechanisms and applications with the required malleability and linkability.

To illustrate how Aleph-Tcl can be used, here are some examples:

RM objects:

```
agent Leandro -name {Leandro Navarro} -picture
~leandro/picture.xbm \ -defaultrole staff -roles {staff comic
pangea}
resource Printer -name {Color Deskwriter at UPC} -location
{Serveis \ Centrals, 2nd floor} -type {PostScript Level 2} \
-url lpr://diable.upc.es:/dev/ttyd03
activity UPC-3-X -context .comic.strand3 -participants leandro \
-class private -description {Writing on S*S and MOI Archs.}
```

Binding events to actions:

```
bind .comic.strand3 <New-Document> {show_awareness [event info]}
```

Other actions (making changes permanent, changing an object, deciding course of action given certain attributes):

```
Printer permanent
Leandro config -defaultrole comic
switch $role {staff - comic {incr perm} default {decr perm}}
```

One subset of the Aleph-Tcl notation (that has been implemented so far) corresponding to the Resource Manager is as follows:

```
obj_name(qualsevol) get -type
Results are: agent, role, resource or context
```

AGENTS:

```
agent obj_name [-new] -name nomusuari
agent -list
obj_name(agent) get -name
obj_name(agent) get -roles
obj_name(agent) put -role rolename contextname
obj_name(agent) delete [-role rolename contextname]
obj_name(agent) put -incontext contextename
obj_name(agent) move [-from contexte1 ] -to contexte2
obj_name(agent) delete [-from contextname]
obj_name(agent) get -context
```

ROLES:

```
role obj_name [-new] -name nom
obj_name(role) put resource [-acco accounting] [-rights rights]
obj_name(role) get -name
obj_name(role) get -rights [+about resourcename]
obj_name(role) delete [-rights resourcename]
role -list
```

RESOURCES:

```
resource obj_name [-new load] -name nom
obj_name(resource) get -name
obj_name(resource) get -policy
obj_name(resource) put -policy load
obj_name(resource) delete
obj_name(resource) put -context contextname
obj_name(resource) move [-from contextname1] to contextname2
obj_name(resource) delete [-fromcontext contextname]
obj_name(resource) get -contexts
obj_name(resource) get -records
obj_name(resource) get -methods
obj_name(resource) put -record recordname
obj_name(resource) put -method methodname,access
path,output,length
obj_name(resource) delete [-record recordname]
obj_name(resource) delete [-method methodname]
obj_name(resource) get -owners
resource -list
```

CONTEXT:

```
context obj_name [-new] -name nom
context -list
obj_name(context) get -name
obj_name(context) get -indexes
obj_name(context) get -hierarchy
obj_name(context) get -below
obj_name(context) get -father
obj_name(context) put -index indexname
obj_name(context) delete [-index indexname]
```

SEARCH and SELECT CANDIDATES:

```
search string/integer name restval [compare] [-objects]
select string/integer name restval [compare] username context
obj_name(resource) connect
```

In the following two sections, the malleability and linkability of the S*S and the Aleph prototype and notation is argumented.

5. Malleability

The malleability feature of mechanisms of interaction has been defined as supporting users in making *global and permanent* changes to its behaviour.

The S*S and the Aleph prototype actors have facilities to change, specify and respecify the behaviour of the mechanisms either directly, by invoking the configuration primitives through the control panel of the mechanism, or indirectly, by doing actions, which in form of events change the behaviour mechanisms as a consequence of changes in their environment. The interpreted nature of the Aleph-Tcl language gives actors a high degree of control of the execution of the mechanism that can be changed dramatically to cope with unexpected contingencies. The part of the Aleph-Tcl, implemented in C language, which corresponds to the objects of cooperative work cannot be changed so easily. These objects are not subject to frequent changes, so this is not an inconvenient.

In addition, designers are specifying the level of malleability when they design the primitives to support changes, and when they decide which parts are going to be malleable (specified in Aleph-Tcl) and which are not (specified in a regular programming language such as C).

The control panel of the mechanism, as well as the Aleph-Tcl shell provide visibility to actors at the appropriate semantic level, independently of the state of any related field of work, to exercise control of the execution and respecify their behaviour. These changes can be done cooperatively, as part of the cooperative effort, while the mechanism is running.

The Tcl language supports radical changes while an application is running. Attributes of an object can be made (`Leandro config -defaultrole comic`), or major changes such as redefining a procedure in the Resource Manager object (`send rm {proc authorize {role resource} { if ...}}`). Which changes must be permitted or not is subject of further study.

Changes will be applicable as soon as they are made. Propagation of changes to other components is done by the event distribution mechanism. It is not clear whether this is the best mechanism for propagation of changes or not. Certainly changes may be either propagated by events or propagated by “lazy” propagation: the next time that the mechanism is invoked.

6. Linkability

MOIs are local and temporal closures [Gerson and Star 86], no single MOI will apply to all aspects of articulation work in all domains of work. The general notation must provide means for establishing local and temporary links between a set of mechanisms (to other MOI, to the wider organisational context) [Risø-3-10].

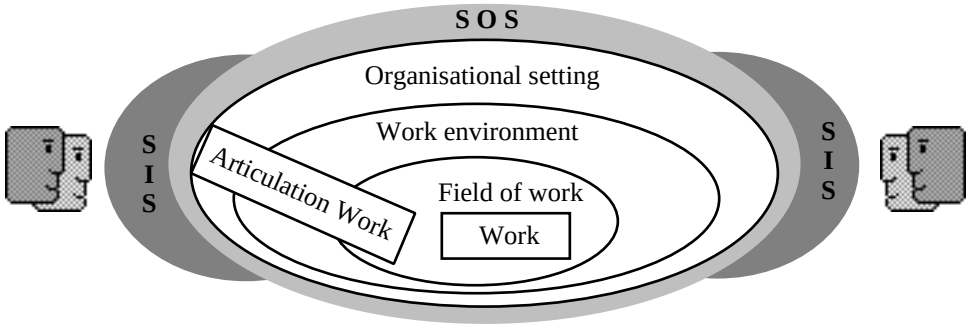


Fig. 6 The S*S provides an environment for applications to do work and to articulate work of primitives at the semantic level of sharing, domain-specific work and articulation work. The Resource Manager, the Event Service, the History Service among others, provide the means to build mechanisms of interaction and delegate most of the burden of articulation work to the appropriate S*S components.

The Resource Manager, with other S*S components provide primitives to support articulation work with respect to most of the objects of articulation work identified in tasks 3.1 and 3.2.

Mechanisms of Interaction specified in terms of the primitives of the S*S can be linked together. Given the examples in [Risø-3-10]:

- Two workflows generated with two different β level notations (graphs and Petri-nets) can be linked by means of the public basic concepts of state and action. These public concepts, among other, can be shared by means of the SOS Sharing Service, and use the Aleph-Tcl notation to link both mechanisms.
- Linking a workflow MOI with a MOI governing the organisational context. The latter MOI corresponds to the Resource Manager, so when the workflow execution reaches a point where an action must be taken, the Resource Manager will receive a request (in the form of an explicit request for permission or as an event produced by the workflow and distributed by the Event Service) to grant permission to an action, and an organisational rule will be evaluated.
- Linking the context of usage of the MOI. The Resource Manager (or the Finder component) provides awareness when resources are used. Two independent mechanisms sharing a resource can be aware that the other is using a shared resource in various ways. For example, one mechanism can explicitly query the Finder to monitor the usage of the shared resource, or it

can express their interest to the Event Service to get Events related to that resource.

In general, the Resource Manager will be responsible for enforcing and applying the organisational policies and restrictions to the objects of articulation work. Any component can use the primitives provided by the Event Service to advertise, supervise, monitor, inform of changes to the components of the S*S. For example, the Resource Manager can produce events as resources are accessed so that it provides the means to support sharing a resource (events that the Locking Manager may receive and transform). Mechanisms can export primitives, values, information to the Sharing Service, and record relevant changes in the History Service.

Given that the Aleph-Tcl notation provides constructions to combine (link) components, to associate behaviour to events, to create new components, to manipulate grammars at γ level (by reconfiguring the Resource Manager entities: adding or modifying a role, rules, actors, resource, etc.) it can be argued that Aleph-Tcl is a γ notation for the S*S, and that a subset of Aleph-Tcl, the subset of domain-specific primitives associated to the visualisation, specification, respecification of a given mechanism, is a β notation for that mechanism. Another β level notation is the graphic notation provided by the control panel of a mechanism that is used to respecify their α notation.

7. Conclusions

We have introduced the S*S architecture and the Aleph prototype in terms of the work on architectures for malleable and linkable mechanisms of interaction and as an environment where both architectures can be put together in one demonstrator. The S*S provides an flexible architecture with primitives at the level of cooperative work, a cooperative interface notation (SIS) and a malleable textual notation (Aleph-Tcl) to develop flexible and linkable mechanisms of interaction applicable on many application domains. A shared goal of both research is to delegate to artifacts much of the drudgery of articulation work; articulate more effectively and with higher degree of flexibility [Schmidt et al. 1993].

The Resource Management primitives provide support for most of the salient dimensions of articulation work in an organisational environment. The way the Resource Manager decides the best candidate and takes into account the complex restrictions and environmental requirements is subject of further study and development. Applications such as workflows, calendar, classification tools provide more specific tools for articulation work. These applications are all interdependent, and they are all based on a common set primitives and a common implicit notation provided by the S*S components. Therefore they can be built and linked very easily to form new mechanisms by means of the Aleph-Tcl (explicit) notation. Given that it is an extensible interpreted language, Aleph-Tcl supports actors in making global and permanent changes to their behaviour: it is

also a malleable notation. Further work is required to use the Aleph prototype in a more realistic situation, increasing the complexity of the rules and requirements that the Resource Manager can support, and adding cooperative applications built on the Aleph-Tcl language.

8. References

GMD-4-5; W. Prinz, J. Mariani, “Awareness of Co-Workers in Collaborative Object Systems”. Comic internal document, 1994.

Simone 94, Risø-3-10; Simone et al. “An architecture for malleable and linkable mechansims of interaction”. Comic internal document, 1994.

Simone 94, Milan-3-2; C. Simone et al. “Mechanisms of Interaction based on conversations”. Comic internal document, 1994.

Herskind 94, Risø-3-13; S. Herskind, H. Nielsen “Designing Mechanisms of Interaction”. Comic internal document, 1994.

Schmidt 93, “Computational Notations of Mechanisms of Interaction: Dimensions, Features and Requirements”. Comic internal document, 1993.

E. Gerson, S.L. Star, “Analyzing Due Process in the Workplace”, ACM TOIS, vol 4, no 3, July 1986.

Malone 92, “Experiments with Oval: A Radically Tailorable Tool for Cooperative Work”, CSCW 92.

Johnson 92; “Supporting Exploratory CSCW with the EGRET Framework”, CSCW 92.

Kaplan 92; “Flexible, Active Support for Cooperative Work with Conversation Builder”, CSCW 92.

Deliv 3.1; Eds. Simone, C. and Schmidt, K. “Computational Mechanisms of Interaction for CSCW”. Comic internal document, 1993.

UPC-4-2; Rodriguez, G. and Navarro, L. “A view of the SOS: refinements”. Comic internal document, 1994.

UPC-4-3; Rodriguez, G. “The design of the Resource manager and the Trader”. Comic internal document, 1994.

ORDIT 93; ESPRIT 2301, 1989-1993 Project Documentation.

ISA 92; APM/RC.422.00, D. Iggulden, Enterprise Architecture, ISA Project, 1992

Ousterhout 93; Ousterhout, J. K., “Tcl And the Tk Toolkit”, URL=ftp://cs.berkeley.edu/book*.

A tool for creating demonstrations of cooperative systems

Tuomo Tuikka, Jouni Kokkoniemi, Tuomo Lalli, Kari Kuutti

University of Oulu

This report describes a tool which can be used to demonstrate cooperative systems. We have named it as MOI-tool. The tool has been built for Apple Macintosh computers in network using HyperCard. The tool has two purposes: research and practical purpose. The research purpose is connected with the primitive operations and their relationship with the notations, describing mechanisms of interaction used in cooperative work. The practical purpose is to support quick and easy creation of demonstrations of cooperative interfaces in multiple workstation. Thus, it should be easy to create demonstrations for example of calendar, flow control or state control applications between many workstations. The demonstrations should work in a way sufficient enough to support discussions with possible users of similar systems, furthermore users should better understand how proper application could help them in their work.

The report is divided into two main sections. First section explains the idea of how tool has been designed, what kind of ideas we have had and what are the conclusions where we are at the moment. Second section (third section) is a manual and an example of how a user can build a demonstration.

1. Introduction

The idea to develop the MOI-tool originated already a few years ago, when the fourth author was working as a consultant in a couple of system development projects. Participatory design in a form or another was used in all those projects, and a considerable number of people without any previous experience about computer systems was involved. During the design process it was found that some features of computer systems are especially difficult to grasp by people who have no prior experience about them. The major problem was in understanding potential connections between different tasks through the system. These connections could be either direct dependencies between tasks or more loose interdependencies, for example when it was possible to monitor the system status changed by other's actions and adapt one's own actions to it. These features were also found very difficult and cumbersome to explain and illustrate. There were tools available for building screen demos and they were adequate for that purpose, but they didn't help in demonstrating a multi-user system as a whole. Then arose an idea to have a simple tool to illustrate and demonstrate features of multi-user systems in participatory design situations. An ideal tool would have been possible to install into a few portable computers, take these to a user site, connect a simple local network between machines and rapidly build, run and modify a multi-user demonstration illustrating the necessary features.

Such a tool was never realized during the design projects, but when COMIC strand 3 ideas on mechanisms of interaction started to take shape, the idea was remembered again, because some of the difficulties encountered in earlier projects had been in illustrating system use in “articulation work” in the strand 3 sense. In order to have a useful tool in that participatory design situation it would have been in fact necessary to be able to build a demonstrator of a computational mechanism of interaction. Thus such a tool could have some relevance for strand 3 work too. When a situation occurred to use cheap resources - a student project - a decision was made to build a prototype.

The idea was discussed in the PMC meeting in Manchester in November 1993. It was obvious that the theoretical work in strand 3 was not yet in the stage where it would have had results to be fed directly into design. So the tool was developed using a time-proven SE method: hacking bottom-up iteratively and using practical scenarios as requirements, against which design ideas were continuously compared. Four different scenarios, each describing a use situation of different multi-user systems, based loosely on the experiences of earlier projects, were developed as starting points, and the goal stated was that the tool to be built should be capable to be used in producing a simulation of all of them.

First sketch of the tool was realized as a student design project which is obligatory for all postgraduate students during their fourth year of study. A group of four students was formed for the project which lasted the spring term -94. The time reserved for the project is nominally 250 hours per student, but it is not unusual that this has been exceeded, as happened in this case as well. The first author of this paper was working as a part-time manager (nominally 1 day/week) for the project. The project produced the first working version of the MOI-tool. Although the first version filled most of the initially stated goals it was not by any means optimal and lots of possibilities in refining both the functionality and especially the user interface were proposed. During the summer 1994 several of these ideas were explored and most promising of them were included in the tool. Nevertheless, the tool is still under development although recent version of it is sent to COMIC partners to be evaluated.

Second section of this report describes MOI-tool development and the background where all has started and third section is an introduction and manual for using the tool. MOI-tool is available from COMIC server and the same stack has a presentation included.

1.1. What is MOI-tool?

MOI-tool is targeted to allow an easy way to create demonstrations for a variety of CSCW applications. The tool is used to build individual screen interfaces for two or more workstations and to link the behavior of different fields, buttons etc. on the different screens in a way to demonstrate certain essential features of some CSCW applications. We have selected four examples for demonstrating functionality of CSCW applications and defined the design of our tool through

these cases. One of those examples is described in section 3 in more detail because we considered it most difficult and challenging. Besides it turned out to have all the properties that were useful for building the other cases.

The MOI-tool is based on HyperCard and it uses correspondingly Macintosh Graphical User Interface elements in interface building. We are still discussing on the functionalities and how they should be formed and now we are sending the tool to 'beta'-testers to get feedback of the ideas. We consider the tool as an aid to be used in discussion about theory of mechanism of interaction, thus it is probably constantly under change. The concepts we have defined and used are mostly in relation to the software environment they are in, i.e. the HyperCard. We have selected HyperCard as a development platform for reasons to be specified later. But first few words of HyperCard and what is the boundary of our approach to 'usual' HyperCard application. Furthermore we will be focusing on the concept of objects and primitives and what we mean by them.

1.2. HyperCard

HyperCard is a software tool that allows you to do more with your computer. In HyperCard, information appears on cards. Cards can contain both text and graphics. One or more cards are grouped together into stacks. A stack is a HyperCard document which can help you do many different things — for example, you could use a stack to keep track of your appointments, manage your expenses, learn a new language, or play music from an audio compact disc. (Domurat, 1991)

There are two object types of plain HyperCard that are the most important for the MOI-tool. They are buttons and fields. Buttons are active areas ("hot spots") on the screen that you can click to make things happen in stacks. For example, a button might take you to another card, show you a picture, launch an application, show an animation, or play music. Fields are containers for field text. You can perform standard editing operations in fields: drag over text to select it, copy or cut it, and paste it elsewhere. (Domurat, 1991)

Adding functionality to HyperCard objects is possible by using HyperTalk®, which is HyperCard's script language. It lets you write English-like statements that respond to events (such as when the user clicks a button or goes to a new card). In HyperTalk, responding to an event is called handling the event. As a scripter, you will write a specific handler for each event that you want your stack to handle. A collection of handlers is called a script. The relation of MOI-tool to HyperCard is that it expands HyperCard's normal objects, buttons and fields, to be "cooperative objects". Buttons and fields are given additional properties through script language which fulfills ideas presented in this paper.

1.3. Objects of user interface and their management

Objects we have here in mind are the objects in the user interface of HyperCard in distinction with objects of articulation work etc. For further introduction of HyperCard environment it is recommended that you read HyperCard help files.

Fields are objects that contain information in either textual or numerical form. With MOI-tool you can add actions with cooperative features to the field by attaching and combining *primitives* to them. By *actions* we mean any kind of functionality added to the object. Actions can be *triggered* in many different ways which are explained later.

Buttons can be used to start actions that affect other interface objects. These actions are launched either by clicking the button with mouse or giving the button object a specific command. Furthermore we have added an object called **Scrolling List** for experimental purposes due to needs in requirements.

Activities that handle objects can and must be accomplished using MOI-tool if cooperative features are desired to be used. These activities are thought to be general considering graphical user interfaces and they are also needed to give proper user control for object handling. We can indicate at least following activities: Create, Define, Change, Destroy. Some of those activities can be done with HyperCard tools and some with our additional tools in MOI-tool.

Create: "Cooperative object" must be created by choosing Toolbox from menu bar and choosing create field etc. from pull-down menu. (Figure 1.3-1). Although objects look like the same when created from normal objects-menu, they include additional properties.

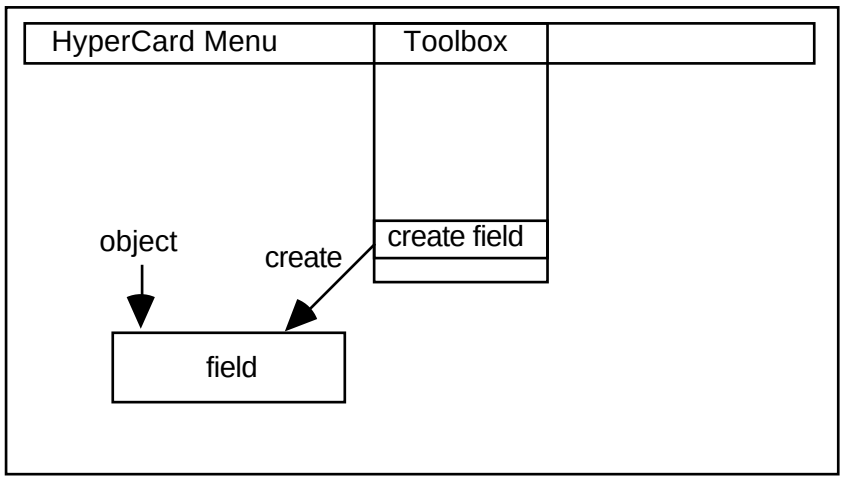


Figure 1.3-1: Create an object

Define: If cooperative features are desired the properties of object must be defined by using ObjectTool which is also our addition to HyperCard. Both primitives and their links can be defined by ObjectTool. You can use ObjectTool by choosing Toolbox-menu and ObjectTool from menu bar. Other properties,

such as button type (radio button, shadowed button) etc. can be changed from HyperCard's own facilities:

Change: The size and other 'external' properties of object can be changed with HyperCard's own tools except for scrolling list-object. Unfortunately, in this version of MOI-tool changes to cooperative features after ObjectTool are possible only from HyperCard's script editor.

Destroy: Objects are deleted with HyperCard's own tools. When deleting an object you must take care that there will not be any tangling links. In other words you may not delete an object that receives information unless you also delete the object that sends information. Otherwise HyperCard gives you an error-message every time you try to use the object.

1.4. Primitives of MOI

We have defined primitives that satisfy the needs of building required actions in almost all of our example cases. In our terminology our set of primitives means such a set of primitive functions implemented inside the HyperCard that can be used to build up all our described required functions for exemplary cases. The choice of such primitives is quite ad hoc, but the main idea is to use them to create functions for building the cases we had. Furthermore, primitive means a function that has no connection to other primitives. This has been problematic and has to be considered more, however. For example there might be a need for a primitive that knows what other primitives have done before. This problem comes up explicitly in the case we introduce later. In the meanwhile we are avoiding this situation by creating primitives that are self-sufficient.

After interface object is created a primitive can be attached to it. Adding more and more primitives to the list forms a *set of primitives* that performs an *action*. Primitives, and actions they form, are launched in different situations, for example when <Enter> is pushed while the cursor is in the field or when the button is clicked. (Figure 1.4-1) Besides a user action, the trigger could be also time or some other primitive which is doing something.

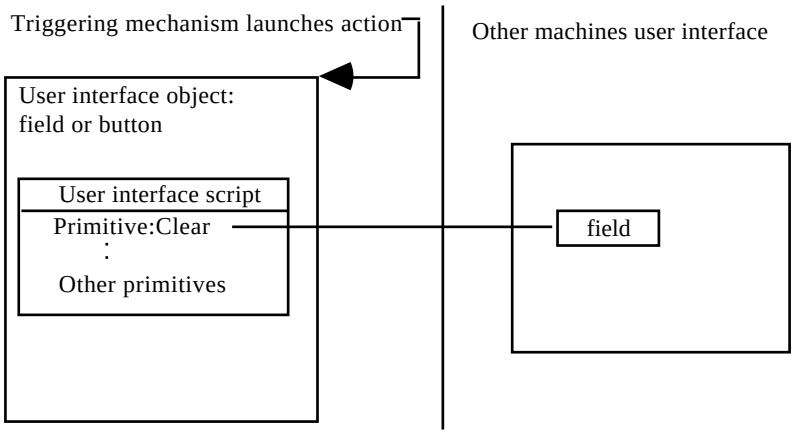


Figure 1.4-1: Primitive clears field in other machines user interface

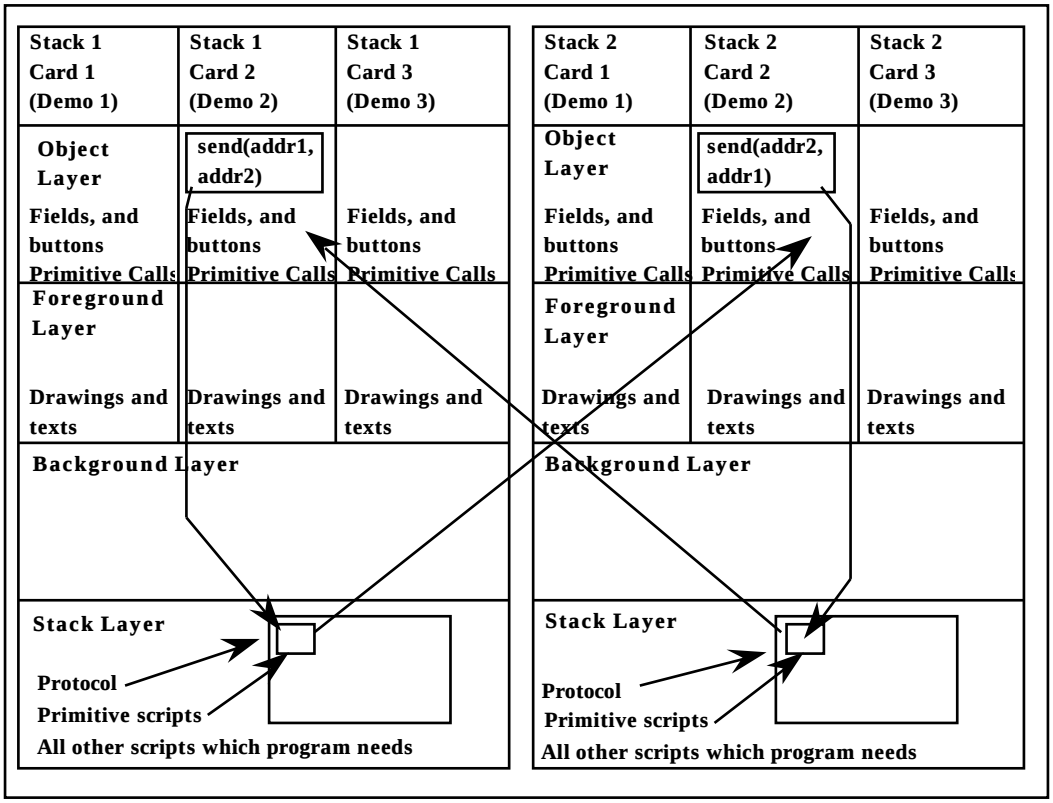


Figure 1.4-2: Principle of program layers

Figure 1.4-2 shows how a primitive call is inside the field-object created with MOI-tool. The script of field has a call to primitive send (or whatever its name is) which is located in the stack layer as all the other primitives. Stack layer handles the message and forwards it to the address in another stack which can be in the same or another machine.

Primitives are executed in the same order that they are attached to the object. Same primitive can be attached to the object several times. We have implemented a first sketch of giving all parameters for primitives in dialog-boxes (ObjectTool), thus it is not necessary to pay attention to the exact format of the primitives. We had also an idea of two types of primitives, the ones using static and the others dynamic linking. These linking types could be compared with absolute and relative references used in spreadsheet programs. In most cases primitives using static links are all you need. Dynamic linking means that if you don’t know where to link e.g. in a field, the program takes care of the linking into a corresponding place. Thus it is possible to demonstrate e.g. timetables. Both linking styles are implemented into recent scripts, but instead of being separate they are in a same primitive.

2. Design of the MOI-tool

There were several design decisions that were made before building the MOI-tool. Architectural design decision depending on the requirements we were given and the problem of what is the approach to this problem when we have four exemplary cases of cooperative systems as a source of innovation. Since this was an innovative software project we thought that prototyping could be a useful way of discussing ideas and trying to find ways to solve our design problems which would rise from four cases.

2.1. Design approach taken

We studied the example cases thoroughly and used descriptions of programs such as The Coordinator, Lotus Notes etc. described in (Schmidt, Simone, Carstensen, Hewitt, & Sørensen, 1993) as a reference and background material for design of our tool. First paper sketch was produced in mid of February ‘94 by brainstorming and a scenario of the user interface was created. Since the original task description of four cases was somewhat simplified we went on by describing a detailed example of all the cases. This way we were able to identify basic functions which were characteristic to the case in question. Because of the nature of HyperCard we started thinking in rather early stage in terms of user interface objects like fields, buttons, cards and stacks. This might restrict our perspectives of user interface, however.

2.2. System architecture

In the early phase of our project we chose Apple Macintosh and HyperCard as software engineering environment. The reason for this was that Apple Macintosh has network facilities included in the hardware and HyperCard is known as being an easy application for creating interface demonstrations. Furthermore we thought our program could be an extension to HyperCard giving it a new dimension with a tool set of creating objects for cooperative work. However, a database application was also considered as a possible aid for us but HyperCard was superior when thinking of user interface manipulation by the user of our application.

2.3. Overall requirements and decisions

First of all we must remember what kind of restrictions we are going to encounter. Computer system and application are restricting factors and the choice of a platform influences further development. Overall requirements for the system were that:

- User interface for the MOI-tool should be easy to use.
- Computer should be portable.
- Network should work with plug and play principle.
- It should be possible to connect objects in different screens to each other.

The work is dependent on restrictions of HyperCard. Thus, the user interface should be consistent with this environment, which consequently affects to the selections of user interface objects with ‘cooperative features’. After defining objects we should define their behavior in selected circumstances. Selecting objects and actions are two main definition tasks because they affect design further on. For instance what are the situations when the objects do something or what are the triggers that start actions. Following bullets gather these issues:

- We should be consistent with the environment selected.
- Objects that we use and which have actions as properties should be selected.
- Actions to be realized should be defined.
- Objects react to an event of some kind and these events that trigger actions in objects should be defined.

Actions as we think of them is a selected behavior of an object. Action is what we want to build with our tool into an object. To make this possible there should be a set of primitive operations that can be used to build an action.

- Primitives that fulfill actions should be defined.

These primitives have at least two criteria they have to fulfill.

- It should be possible to combine primitives to fulfill actions.
- Primitives should be independent so that information is not changed between two or more primitives. That is why we call them primitives.

As a conclusion we could say that this is not an easy task and needs a lot of discussion during the design phase and many questions arise. For example, how can we create all primitives optimally? Is it possible to create primitives in this sense at all? Of course our first presumption is that it should be possible to use a set of operations i.e. *primitives* that perform the actions when they are combined. Furthermore, is it possible to use primitives for creating also actions that are not defined? Are there situations where primitives are dependent on the state of the demonstration? What are the triggers that start actions? All comments and suggestions are welcome for further development.

2.4. Cases

Our goal has been to achieve the creation of four example cases given for us by using our system. This view of four situations of cooperative work cases surely is quite restrictive considering cooperative work or cooperative interfaces but in a limited time schedule the design was thought to be possible. We must remember that all the cases were given and that not too much time was used in analyzing other systems. The original cases were described in the following way:

Case 1. In this case we should be able to simulate a document handling system. The system should register documents, their dates when they arrived and who has them. Archivist should be able to see where a document is and if someone would send it further, this change would appear to the screen.

Case 2. We should be able to simulate a common calendar. If somebody reserves certain hours, for example for a meeting room, this change would be visible in other screens.

Case 3. Project management simulation where project has netlike abstraction of its tasks and tasks are allocated to different persons. For example tasks that user can do or possibly all the tasks waiting for my tasks to be done could be visible.

Case 4. This case is a demonstration of a timetable system for a medical organization. Office personnel should be able to take reservations to several physicians and organize them leaving space for urgent patients. When patients are treated or they do not come the timetable changes for each of the physicians. Also office personnel should be aware of changes so that they can add new patients or remove old reservations. Of course the change to timetable should be seen by the physician.

2.5. Case 1 as an example

In the beginning of this section we will describe how our first case situation was defined and what kind of actions it has. Explanation in this paper tries to be as elaborate as possible because primitives are derived through similar discussion considering all the cases. Case 1 was chosen to be presented here since it was the most complex system that we had. We expect other cases to be a subset of this case. Furthermore, this case was brought not only as an example how we could make more use of a scrolling list-object but also illustrating the problem of dividing primitives. The questions that arise are not resolved but thought to be possibly as general problems we might encounter in other more complex cases than this.

While reading this description one has to remember that our ideas have been generated mainly using brainstorming while thinking how the system works. Some of the presented primitives are not implemented yet and possible will not be. It could be useful to rethink usefulness of the examples but the approach we have taken gains from its simplicity.

2.5.1. Description

Our first case example was a document handling system of an imaginary organization. The system is used by archivist and people who handle documents. Archivist has a user interface of a system which shows a list of documents where all items have day of arrival and where the documents have been sent (“to whom”-field). Other users have an interface which shows information of documents that belongs to the user. Figure 2.5.1-1 is an overall picture of how the system is thought to be working. In figure we have presented three workstations working together. Hence, archivist, user 1 and user 2 have a workstation each. Let’s assume that one row in the picture is part of a list of documents in an application that handles document registration. This one row shows then

information of one document, which is thus listed in the archivists list and in the list of whoever has it.

The *archivist* takes care of filling the document list in his user interface and other users should in principle inform the archivist if they receive or send a document. When the archivist receives a document or information that a document has arrived the user must add manually the name or identifier of document into column “document” and day of arrival into column “date” in the list of documents. If the document is not sent further “to whom”-field is not filled. Otherwise the document is sent to some other user and the name of user is inserted into corresponding field of column “to whom”.

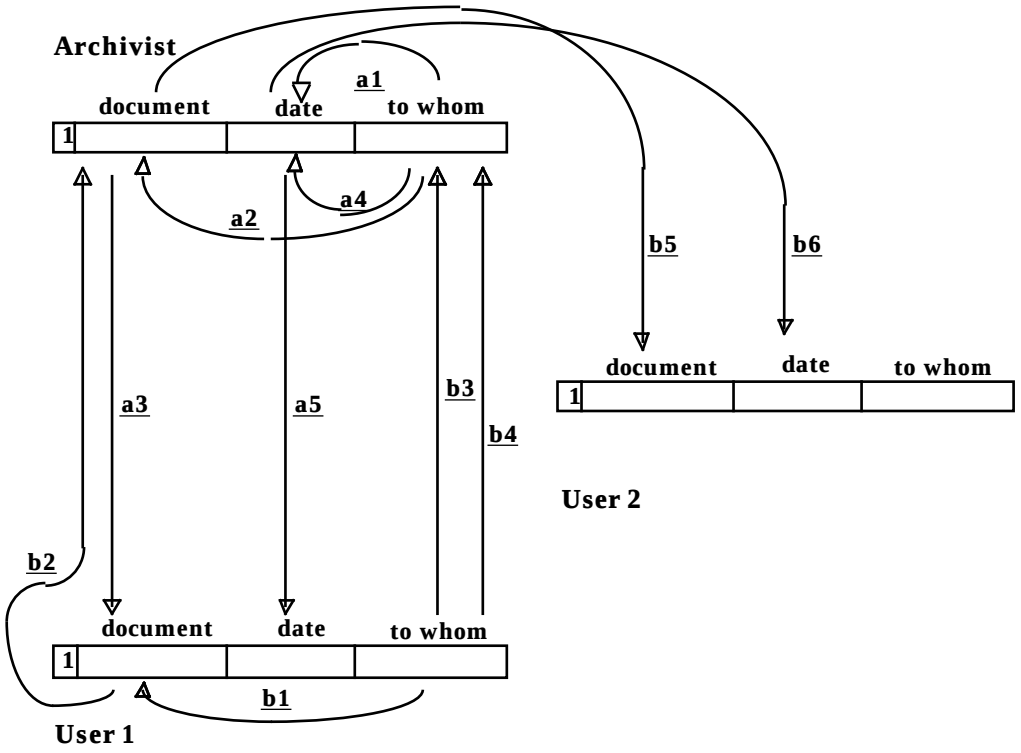


Figure 2.5.1-1: Actions and links

An example illustrates actions best. Presume that archivist sends document to user 1 (manually). In the system, archivist first adds row 1 to the list in user interface. When “to whom”-field has been filled archivist presses enter. After enter things start happen. Date is updated to “date”-field showing when document was sent (action a1). In this case the arrow in the figure 2.5.1-1 shows the originator object for action and arrowhead the destination of the action. To help in understanding this remember that user interface object “to whom”-field is thought to have all of the operations needed to create all these actions.

Actions a2 and a3 happen because we want to send contents of field “document” (see a2) from archivist to user 1’s row which identifies document-name (see a3). Also “date”-field is copied to user 1. Arrow a4 and a5 describe the idea of sending the contents of “date”-field of archivists list to “date”-field of user

1’s list. Now we have described all the actions that are needed for input of a list and copying that information to another machine. It is not difficult to imagine that *complexity* of linking grows after several identified links.

When user 1 wants to send a document to user 2 he must add user 2’s name into column “to whom”-field in his own interface and press enter. B-arrows show how user 1 updates the list after sending document to user 2 manually by writing information “to whom”-field. What happens is that after pressing enter, the field in archivists document row and user 2’s row are updated to show the corresponding situation. Also user 1’s row is cleared. More deeply described b1 and b2 mean that we need the document information to be able to send to the right destination. Contents of “to whom”-field is sent from user 1 to archivist (see b3). Next thing is that specified row in user 1’s interface is cleared. We still have a problem at this point of showing the same information in user 2’s interface. This can be taken care of by sending a command to archivists system which starts the procedure of sending the information to user 2’s system (b4, b5 and b6).

It has been clear now for a while that in this case we are handling a list structure which could raise some problems we are going to discuss later. As we can see archivist has a list of all the documents while other participants have a list their own document information. Presumably we can add the document information in the first available position in receivers list.

2.5.2. Primitives derived from the case 1

First, a list of actions was created. With other examples as a background criteria this list shows the use of primitives we had earlier defined. Due to our experimental approach we were not worried of any problems at this stage. Table 2.5.2-1 introduces the idea by explaining how a primitive could be connected to an action.

action	explanation
a1	Primitive “date” puts new date into column “date” at line 1 .
a2	Primitive Send copies contents of line 1 of a column
a3	“document” to User 1’s workstation into first free line of column “document”. Line 1 of column “to whom” in archivists interface specifies the user whom to sent.
a4	Primitive Send sends column “date” from line 1.
a5	Contents of column “date” is copied to User 1’s workstation into first free line of column “date”. Line 1 of column “to whom” of archivists interface specified the user whom to send.

Table 2.5.2-1: More elaborate explanation of a-codes

Thus, we can see that we need at least primitive for putting *date* to a field and a primitive that *send*s information from one field to another. In the latter case the functionality of the primitive reminds of copying contents of field to another. Which it actually does while the originator field is left intact. Otherwise if

originator was removed it would mean that we move contents of field to another field. This is one functionality more, however. Function for *clear*ing the originator field in send could be useful. Clear could also be useful somewhere else too and that is why we have thought of separate clear.

action	explanation
b1	Primitive exam examines column “document” and gets data for comparing.
b2	Primitive exam examines location of document in archivists list of documents and returns this location.
b3	Primitive Send sends contents of column “to whom” in line 1 to archivists workstation into line (specified by exam) of column “to whom”.
b4	Primitive Launch launches archivist column “to whom” in line (specified by exam). Launch starts executing of primitives which are inside archivists column “to whom”-field.
b5	Primitive Send of “to whom”-field in line 1 of column “document” starts working. Contents of line 1 of column “document” is copied to User 2’s workstation into first free line. “To whom”-field of archivists interface is now “user 2” and it specifies the address for Send.
b6	Primitive Send sends contents “date” field to User 2’s workstation into first free line of column “date”. “To whom”-field of archivists interface is now “user 2” and it specifies the address to be sent.

Table 2.5.2-2: More elaborate explanation of b-codes

Primitive candidates

As we can see *exam*, *send*, *date*, *launch* and *clear* are probable candidates as primitives. From earlier studies we have decided that date, launch, clear and send should be realized. Their parameters and location are quite simple: date(where to put date), launch(where from), clear(what). Thus they have only one address and they can be attached almost anywhere. After implementing these primitives to be able point at field object (address) also versions that understand scrolling object with multiple lines were created.

Send is an interesting primitive here and definitely the most used. It needs usually only two parameters as addresses in the case of two fields, where from and where to. Both in a2,a3 and a4,a5 send-primitive examples need at least 2 addresses: where from and where to send, this is trivial. But what if we must search for an empty row, or we have to do any kind of search. This is a question that of course depends on the requirements. Since we are not trying to create a database program with relational database functionality we apparently must also try to restrict this development of the primitives in MOI-tool at some stage. Nevertheless, send has different modes of action in the case which was presented; sometimes send needs to know what comes from exam and sometimes not, also “to whom”-field has been used as an address for sending. It is probable that we need a primitive with many parameters or several connected primitives to handle this send. The latter problem of using “to whom”-field as an address was solved with presenting new primitive called aliasSend. This primitive is attached to the

field which calls aliasSend and it recognizes the workstations names. The problem of examining and sending is discussed more in the summary.

2.6. Summary

The summary of the design selections and state of the program are issues reported in this section. All the selections of primitives and their properties are listed here as we have built them by now (end of August ‘94). The last section discusses about problems we have had and tries to find generalizations of them for further development.

2.6.1. Objects and their behavior

Selecting user interface objects was fairly simple since HyperCard allows us to use at least **field** and **button**-objects, which were the first user interface-objects (UI) we thought to be useful. Fields and buttons created from MOI’s ToolBox-menu include more functionality than ‘normal’ objects: they can be used to link to objects in other workstations. The distinction between a UI-object which has primitives and a UI-object which is only an object for primitive actions is basic for functionality of demonstration. One UI-object can have all the actions needed for the success of demonstration. Furthermore, some actions toward HyperCard’s **card**-object has been found useful for demonstrations.

Scrolling list is an experimental object which we thought to be useful for demonstrating tables etc. It does not have all the properties that objects like fields have since it is an agglomeration of different objects. User can define the size of scrolling list as columns and rows but it doesn’t have any more flexibility. Primitives are usually attached to field objects inside the scrolling list. Primitives objected towards Scrolling list make use of rows in the fields.

Triggering events are mostly dependent on the construction of HyperCard. For field-objects we must use at least two events: enter is pressed (not return) or mouse is clicked outside the field. At least these events cause primitives to run in a field-object. However, it would be also helpful to be able to launch primitives when necessary. Thus, we have implemented a primitive which can be attached to any UI-object to launch a primitive sequence in any UI-object.

In button-objects primitives are triggered every time when mouse is pressed. Launch-primitive is also available for this object. There could be also a trigger when something happens or time-based trigger, but they have not been implemented yet.

2.6.2. Primitive list and properties

Table 2.6.2-1 presents a User Interface-object (UI) available for primitive attachment, its definition in the sense we see it and the primitives that can be pointed to it. Listed primitives can be inside of any of the object's script. All of the primitives and their use has been implemented into a demonstration program available at COMIC-server.

Object	**Definition**	**Primitives possible for pointing at Object**
Field	Field is a container for field text. You can perform standard editing operations in fields: drag over text to select it, copy or cut it, and paste it elsewhere.	Clear Date Time Launch Lock Send hideObject showObject
Button	Buttons are active areas ("hot spots") on the screen that you can click to make things happen in stacks.	Launch
Scrolling list	Scrolling list is an object which has a column (field) of numbers and several fields depending on how users have defined the list. Scrolling list differs from field while having a field with line numbering and tools for scrolling. In this version of MOI-tool (1.0) scrolling list must be created as first object if it is used.	Clear Date Time Launch Lock Send aliasSend Delete Insert
Card	Card is the basic unit of information in a stack. A card is a rectangular area that can contain specific buttons, fields for text, and graphics.	GoToPrev GoToNext

Table 2.6.2-1: Definition of UI-object and primitives possible to point at them

Primitive	Pointed at Object	Behavior at Object
Clear	Field	Clear empties the field
	Button	Clear can be attached to button for clearing an field object. This operation of attaching an primitive is simple because the primitives can be attached to any object in question. *) This will not be mentioned later on.
	Scrolling list	Clear empties a line of Scrolling list
ClearScrFld	Field	Connection not suggested
	Button	Connection N/A. Attachment suggested.
	Scrolling list	Empties one of the scrolling list fields
Time	Field	Time is set to a field
	Button	*)
	Scrolling list	Time is set to a line of scrolling list.
Date	Field	Date is set to a field
	Button	*)
	Scrolling list	Date is set to a line of scrolling list.
Launch	Field	Launches primitives in field script.
	Button	Launches primitives in button script.
	Scrolling list	Launches primitives in field pointed from Scrolling list.
Lock	Field	Locks text field. Makes input to a field impossible.
	Button	*)
	Scrolling list	Locks text field of Scrolling list. Makes input to a field impossible.
Send	Field	Field receives data or data is sent from field to another field leaving text intact in the original field.
	Button	*)
	Scrolling list	Field receives data or data is sent from field to another field leaving text intact in the original field. Send can be used to create ‘transparent’ Scrolling list, since it recognizes the rows of fields.
hideObject	Field	Object is hided.
	Button	Object is hided.
	Scrolling list	One of scrolling list objects is hided.
showObject	Field	Object is shown.
	Button	Object is shown.
	Scrolling list	One of scrolling list objects is shown.

Table 2.6.2-2: Definition of behavior of primitive when attached to an object

Tables 2.6.2-3 describes primitives that are special for scrolling list and are used in other objects only as attachments. Tables 2.6.2-4 describes primitives for handling cards.

Primitive	Pointed at Object	Behavior at Object
Delete	Field	Not available/No attachment suggested
	Button	Attachment of this primitive is suggested to be done only to button object, since it's special behavior.
	Scrolling list	Deletes one specified row of Scrolling list bringing trailing rows upward one row.
Insert	Field	Not available/No attachment suggested
	Button	Attachment of this primitive is suggested to be done only to button object, since it's special behavior.
	Scrolling list	Inserts one row of Scrolling list pushing trailing rows downward one row.
aliasSend	Field	Not available/No attachment
	Button	Not available/No attachment
	Scrolling list	Sends data from given address to the same address in other workstation. Alias name of the workstation is expected to be in the field where this primitive is attached.

Table 2.6.2-3: Definition of behavior of primitive special for scrolling list

Primitive	Pointed at Object	Behavior at Object
GoToNext	Field	Not available/No attachment suggested
	Button	Attachment of this primitive is suggested to be done only to button object.
	Scrolling list	Not available/No attachment suggested
	Card	Changes card to next card in order.
GoToPrev	Field	Not available/No attachment suggested
	Button	Attachment of this primitive is suggested to be done only to button object.
	Scrolling list	Not available/No attachment suggested
	Card	Changes card to previous card in order.

Table 2.6.2-4: Definition of behavior of primitive special for card-object

Primitive	Parameter(s)	Parameter explanation
Clear(address)	address	Clear-primitive is used to clear the contents of field defined by parameter address1. Thus address1 determines the field to clear.
ClearScrFld(address)	address	Address points to scrolling list field which needs to be emptied.
Time(address)	address	Primitive is used to write current time to the field determined by address1. The time is obtained from operating system's time.
Date(address)	address	Date-primitive is used to write today's date to the field determined by address1. The date is obtained from operating system's date.
Launch(address)	address	Launch-primitive sends a launch command to an object determined by address1. Launch starts from executing all the primitives attached to object.
Lock(address)	address	Lock-primitive is used to lock the field determined by address1, so that field's contents cannot be changed.
hideObject(address)	address	Hides object defined by address.
showObject(address)	address	Shows object defined by address.
Send(address1,address2)	address1	Send-primitive sends information from address1 to
	address2	destination address2

Table 2.6.2-5: Primitives and their parameters explained

Delete(address)	address	Primitive empties the row determined by address and moves trailing rows one field upwards. Implemented for button object. Asks for row to be deleted.
Insert(address)	address	Primitive inserts new empty row into location determined by address. Implemented for button object. Asks for row to be inserted.
aliasSend	address	Sends data from given address to the workstation given in the field where the primitive is located. Data will go to corresponding field/line in the other workstation.

Table 2.6.2-6: Primitives and their parameters specially for scrolling list.

GoToNext(address)	address	Address specifies the workstation from which card is changed forward.
GoToPrev(address)	address	Address specifies the workstation from which card is changed backward.

Table 2.6.2-7: Primitives and their parameters specially for scrolling list.

The design has been implemented mostly as stack level scripts in HyperCard so that code is in the same place and easy to read. Therefore all the primitives are coded at stack level and are easily modified using HyperCard script editor. Commenting of the code has been left out (!) because HyperCard has limitations for the size of the scripts.

2.6.3. Problems and discussion

As a lesson of this work we one of the questions we can ask is: How should primitives be selected? One criteria in this kind of experiment could be the amount of primitives i.e. just don’ t care and build one when you need one combine them if there are too many. This is what we have done until now. But if we are going to continue this work a more specific set of primitives and their connections and behavior should be defined.

Table 2.5.2-2 shows the problem of comparing information while sending information into certain position. Therefore primitive exam has been introduced in the explanation. There is a situation where exam returns information to send which means that the primitives are not independent. One important question that arises and expresses the whole process of development: Should we create a new primitive which has all these properties (exam and send) included or should they be separate? Consequently, we would have a primitive that is complex with two clear separate actions: examining something with a criteria and sending something according to the result of examine. If this primitive was not done, there should be a well defined mechanism for changing information between primitives even with other primitives which need information from exam. This would violate the rule we had in the beginning of making the primitives independent. Perhaps they would not then be primitives at all. A common problem has been how to implement a primitive first in the sense that where it is attached. This can be seen at least from aliasSend-primitive, which expects to be attached to an object from where it is called.

One interesting issue for further studies would be adding conditional behavior to objects and primitives. That would mean expanding ObjectTool with a functionality to add limitations and conditions to demonstration. Another idea for further studies would be building mechanism of interaction like studied for example in (Borstrøm, Carstensen, & Sørensen, 1994), (Herskind & Nielsen, 1994) and requirements studied in (Carstensen, Tuikka, & Sørensen, 1994) to

elaborate further the requirements for the tool from social mechanisms of interaction.

What comes to HyperCard it is more and more obvious that HyperCard's limitations restrict the development of the tool. Script editor's limitations and problems like odd invisible characters to code are annoying and impossibility to access some features of HyperCard is restricting. There are also some problems with HyperCard when handling Apple Events. The program sometimes loses messages, and with bad luck gets stuck. For example changing card must be done to other workstation first otherwise primitive GoToNext doesn't work. In this case proper tools for debugging line between workstation would be useful.

3. Using MOI-Tool to create a demonstration

This section explains how MOI-tool can be used to create a demonstration. After reading you should be able to understand how MOI-tool works and use it to create a demonstration. First section instructs with setting up the demonstration building session. Second presents the application environment used and...

3.1. Organizing MOI-tool

We suggest that you use two Apple Macintosh computers with at least 4 MB RAM and HyperCard program version 2.1, Apple LocalTalk network or Ethernet. System should be equal or better than 7. The setting of these operations is described in Macintosh manuals.

After fixing the environment and opening MOI-tool there are some preparations that must be done in order to make primitives work correctly. All participating workstations must be on and HyperCard program of all workstation must be open. There are two stacks you need (on your both machines): DemoWindow and ObjectTool (Figure 3.1-1). Remember to copy both stacks to all of the workstations in use. MOI-tool is built into these stacks, they form the basic environment, everything needed is inside them. If network facilities are in order, like program linking allowed, the stacks are ready to run.

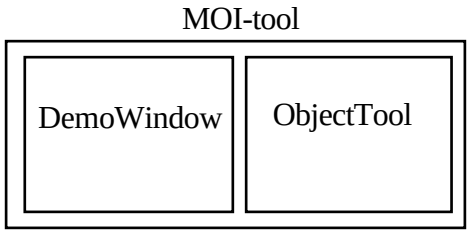


Figure 3.1-1: MOI-tool parts

The most minimal demonstration with MOI-tool can be done by using only one workstation. In this case you need to start two HyperCard versions with different names and open the stacks needed for your demonstrations. Two workstations must be selected, however, but here it means selecting different versions of HyperCard. Thus, it is possible to create demonstration in one workstation, of course ‘cooperative interface’ can be made with two workstation setup. The only difference is then that instead of choosing the same machine in the beginning, choose another Macintosh in the second round of dialog boxes. The sequence is very important because Macintoshes have relative addresses according to the steps in the dialog boxes of the start. REMEMBER ALWAYS GIVE THE NAME OF COMPUTER YOU ARE WORKING ON FIRST, so you won’t get into troubles next time if you change the computer. When changing to another machine just do the same there: the machine on which you are at is given the first name.

3.2. DemoWindow-stack

The main purpose of DemoWindow-stack is to offer a foundation for building your own demonstrations. It has almost all the scripts needed for handling user actions as stack script. You can easily take a look at the script by choosing Stack Info from Objects-menu. Some of the scripts are in other objects although we have tried to minimize coding in them. Furthermore Demowindow-stack has access to all the facilities for creating a demonstration with connected objects. These facilities are collected into a menu which we have named as ‘Toolbox’ for the moment. Toolbox-menu has following items: “New Demo”, “Create Button”, “Create Field”, “Create Scrolling List”, “ObjectTool”, ”Show Selection” and “Show Object Data”.

“New Demo” is used to create a new demonstration card in Demowindow-stack. This can be done also by using HyperCard’s facilities to create a card since ToolBox script does not do anything special except gives the card a name. Take care that you use the naming convention if creating a card in HyperCard’s way. >>However, the card is always situated to the end of stack, which is different from HyperCard.

“Create Button”, **“Create Field”** and **“Create Scrolling List”** are used to create objects into which primitives can be attached. Objects created normally with HyperCard’s Object menu will not work since they don’t have the properties for linking to other workstations.

After selecting one of these menu options either objects or dialog boxes appear. If objects appear and you want to link them to other user interface, it is REQUIRED to select the object first and then start operations with “ObjectTool” to define collection of primitives and object linking.

“ObjectTool” opens ObjectTool-stack which is discussed later.

“Show Selection” shows a small window with the name of the selected object. This is handy when you want to know what is selected for linking.

“Show Object Data” shows a window with primitives attached to all of the user interface objects. It shows also all kind of other information like addresses which are linked into a primitive. “Show Object Data” shows also data of objects which are not existent anymore.

3.3. ObjectTool

Following figure (3.2.1-1) presents overall description of how all primitives are attached to an object using ObjectTool. First, object is chosen by clicking it. You can use menu item “Show Selection” of Toolbox-menu to check which object was selected. After that ObjectTool can be chosen from the Toolbox-menu.

In the first appearing dialog (1) user can choose the primitive to be attached from a list and then press the LINK-button. Depending on the primitive chosen, application presents necessary dialogs (2). After all the required parameters has been given, first dialog is returned and user can attach more primitives to the object (3). When everything is ready and all the primitives are selected, OK button is pressed. During this stage MOI-tool creates the links for the selected primitives (4) without user noticing, however.

Unfortunately, the primitives in the object cannot be changed after this procedure unless the user knows how to use HyperTalk scripting language. Hopefully this can be fixed in later versions of MOI-tool.

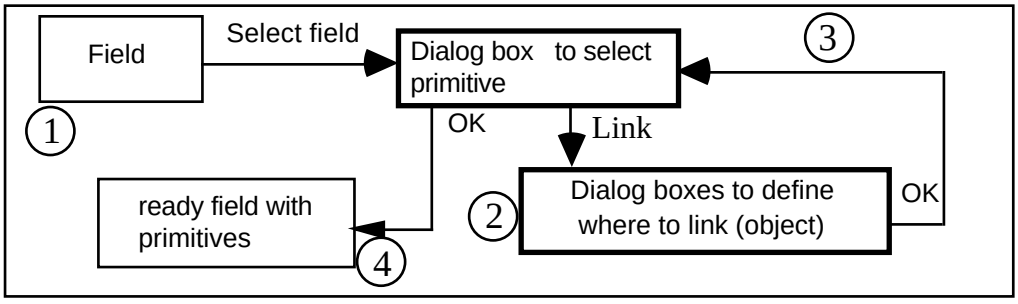


Figure 3.2.1-1: Description of the procedure to attach primitives to an object

ObjectTool contains three different kind of dialogs. In ‘Select primitives for object’- dialog user can choose primitives to be attached to object, in ‘Select address of first (and second) workstation’ and ‘Select address of first (second) object for linking’ dialogs user can choose addresses and objects for primitive attachment.

3.3.1. ‘Select primitives for object’-dialog

‘Select primitives for object’-dialog (figure 3.3.1-1) is the first dialog of ObjectTool. In this dialog user can choose desired primitive from the List of primitives.

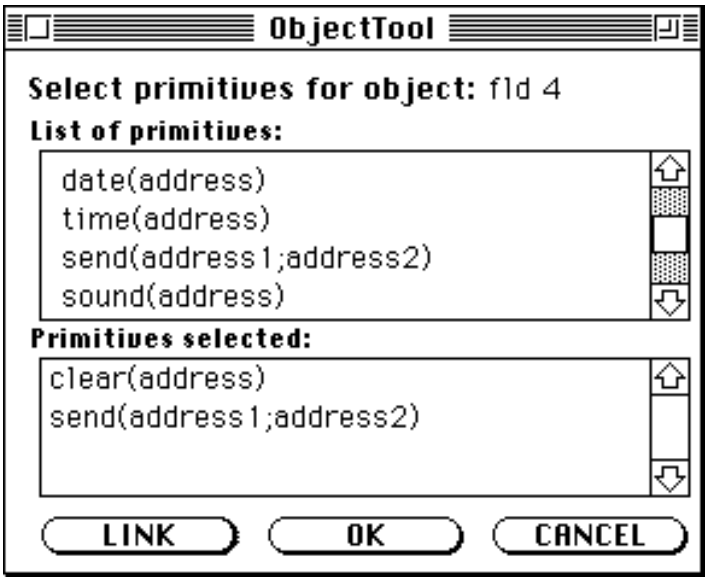


Figure 3.3.1-1: ‘Select primitives for object’-dialog

This dialog contains the following fields and buttons:

“Select primitives for object” shows the name of the object which is under manipulation.

“List of primitives” shows a list of all the primitives that exists. Primitive can be chosen by clicking it.

“Primitives selected” shows the primitives in the order in which they are chosen.

“LINK” opens up a new dialog through which chosen primitive gets necessary parameters. LINK-button is not active unless primitive has been selected from field “List of primitives”. Using LINK-button after selecting a primitive is compulsory.

“OK” accepts all the choices made and user is returned to DemoWindow-stack.

“CANCEL” interrupts using the ObjectTool. No primitives are attached to the object.

After all the selections are done and you can see selected primitives from “Primitives Selected”-field. You can either accept them or cancel the whole operation. If you press “CANCEL” the linking operation is canceled and no primitives is attached into the object. Pressing OK preserves all the primitives and finishes linking.

3.3.2. ‘Select address of first workstation’-dialog

‘Select address of first workstation’-dialog (and second) appears (figure 3.3.2-1) to the screen when a primitive is chosen in the previous dialog and LINK-button is pressed. In this dialog user can choose the workstation to be linked for selected object and primitive.

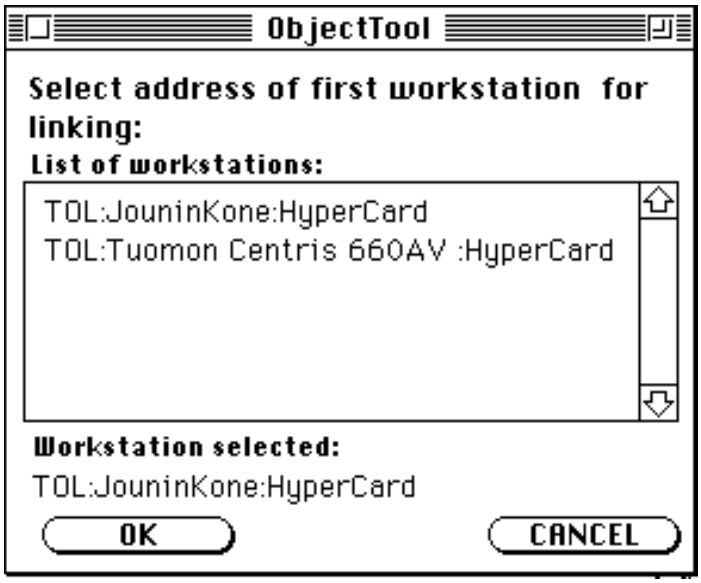


Figure 3.3.2-1: 'Select address of first workstation'-dialog

This dialog contains the following fields and buttons:

"List of workstations" lists all the workstations that participate to the demonstration. User can choose the workstation by clicking the appropriate line.

"Workstation selected" field shows the selected workstation.

"OK" continues to next dialog.

"CANCEL" interrupts the ObjectTool. No primitives are attached to the object.

3.3.3. 'Select address of first object for linking' -dialog

In 'Select address of first object for linking' -dialog (or second) objects to be linked for selected primitive are chosen (figure 3.3.3-1). Dialog appears on the screen when a workstation is chosen in the previous dialog and OK is pressed.

This dialog contains the following fields and buttons:

"Object with primitive" field shows the name of the object of which primitive is about to be attached.

"Workstation selected" shows the name of the workstation selected in the previous dialog.

"Existing objects in workstation" shows a list of all objects on the selected workstation. Object is selected from the list by clicking it.

"OK" saves the choices made for the primitive and returns "**Select primitives for object'**-dialog

"CANCEL" interrupts using the ObjectTool. No primitives are attached to the object.

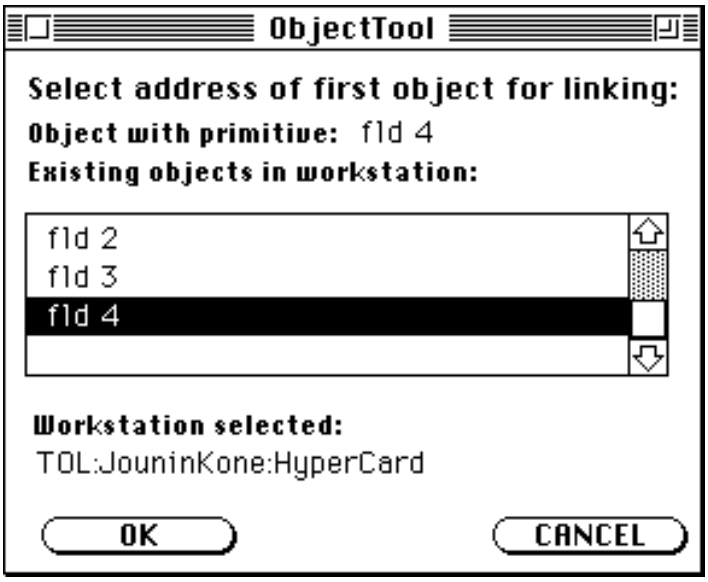


Figure 3.3.3-1: ‘Select address of first object for linking’ -dialog

3.4. How to create a demonstration

This section presents an example of creating a demonstration. First how to open the MOI-tool, then some easy examples of use of fields and buttons and at last an example how to create demonstration of case 4 with scrolling lists.

3.4.1. Opening MOI-tool

Starting DemoWindow-stack brings up the dialog for giving the number of workstations in demo session (figure 3.4.1-1). At least number from two to three workstations can be attached in this dialog box and current workstation is included to this number. All participating workstations must be on and HyperCard program of workstation must be open. Choosing “Cancel” disables linking to other workstations you can browse and modify the stack like normal HyperCard-stack.

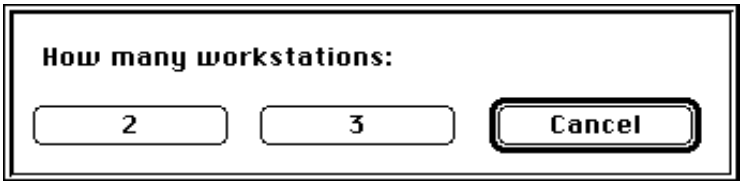


Figure 3.4.1-1: ‘How many workstations’-dialog box

Information of workstations is asked next. This information is actually the address to program connected to the session. First of them is our own address which you can see in figure 3.4.1-2. Select the right zone, machine and program. Press OK.

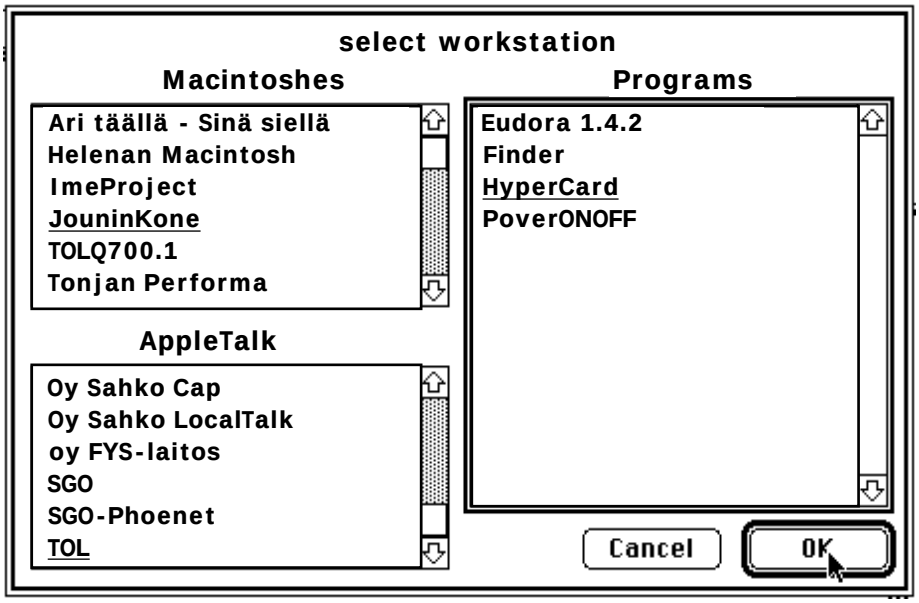


Figure 3.4.1-2: Selecting address of our own workstation

Then alias name for our workstation is asked. In figure 3.4.1-3 we can see this dialog box. Write the alias name you want and press OK. Alias name is used as a workstation name in some cases. This is definitely easier way than giving the real name of the workstation when it is needed.

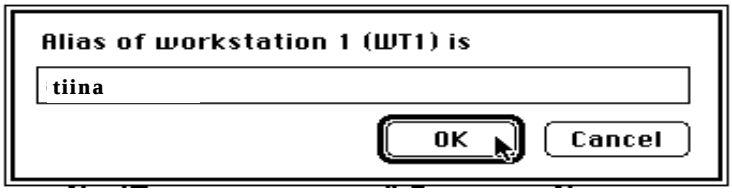


Figure 3.4.1-3: Giving alias name for our own workstation

Other users addresses are given in the same way (figure 3.4.1-4). Press OK (1) and give alias name which in this example is “pekka” (2). Correspondingly: Tuomon Centris/TOL/HyperCard (3) and alias name “Tuomo” (4).

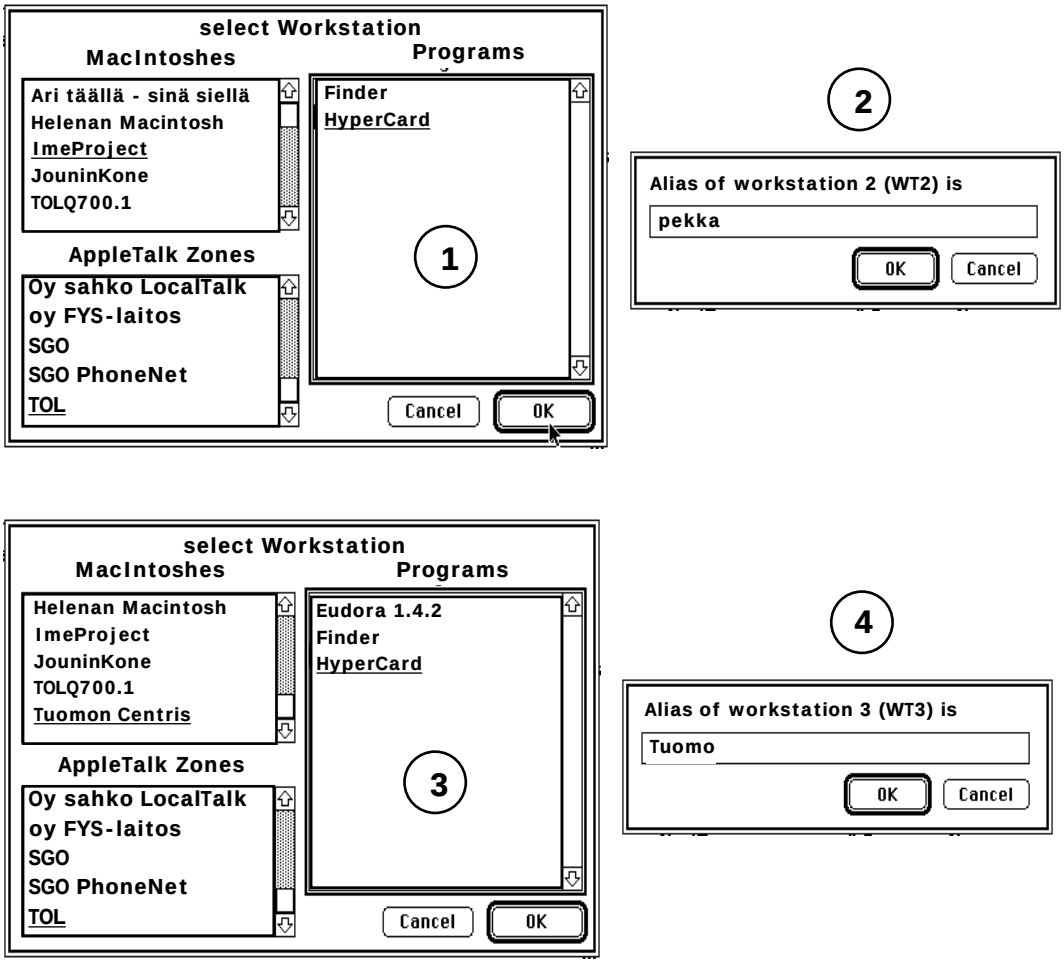


Figure 3.4.1-4: Selecting addresses of other users

Now addresses and alias names should be correct and building of demonstration can start. Build a new demonstration by first creating new card. Select menu item “New Demo” from “Toolbox”-menu. This selection creates new card and names it “demo n” where n is the number of demonstration.

3.4.2. Simple example for linking field-objects

When new demonstration has been created we can add objects and attach primitives to them.
*Create two fields by selecting menu item “Create Field” of menu “ToolBox” to both workstations. Fields in workstation “JouninKone” (or your own workstations) are “fld 1” and “fld 2” and in workstation “ImeProject” they are “fld 3” and “fld 4”, for example.
*Create buttons “btn 1” and “btn 2” to workstation “JouninKone”.

It is possible now to link a field to another field in our own workstation or field in our workstation to field in another workstation or we can use buttons to trigger fields for sending. In the last case fields can be without primitives since button primitives do the work.

Let's assume that we want to attach primitive send to field "fld 1" of workstation "JouninKone" (or of course your own workstation). What primitive does is that it sends data from "fld 1" to "fld 2" of "JouninKone", that is in the same workstation.

*Select "fld 1" by clicking it. You can check selection by opening Show selection window.

*Open ObjectTool. Do the selections shown in table 3.4.2-1

	dialog box	**selection**
1	Select primitives for object	send(address1;address2)
2	Select address of first workstation for linking	TOL:JouninKone:HyperCard (your machine)
3	Select address of first object for linking	fld 1 (for example)
4	Select address of second workstation for linking	TOL:JouninKone:HyperCard (your machine
5	Select address of second object for linking	fld 2 (for example)

Table 3.4.2-1: Primitive send sends data from "fld 1" to "fld 2"

Sending contents of "fld 2" of workstation "JouninKone" to "fld 3" of workstation "ImeProject" can happen at least in two ways. One way is shown in table 3.4.2-2 where button "btn 1" is used.

*Select first button "btn 1"

*Open ObjectTool and attach primitive into selected object in following way.

	dialog box	**selection**
1	Select primitives for object	send(address1;address2)
2	Select address of first workstation for linking	TOL:JouninKone:HyperCard (your machine)
3	Select address of first object for linking	fld 2 (for example)
4	Select address of second workstation for linking	TOL:ImeProject:HyperCard-II (your other machine)
5	Select address of second object for linking	fld 3 (for example)

Table 3.4.2-2: Data is sent from "fld 1" of "JouninKone" to "fld 2" of "ImeProject", "btn 1" is a trigger.

Actually the other way is also shown in table 3.4.2-2. Only difference is that instead selecting "btn 1" first, select field "fld 2" as a target for ObjectTool.

Next we show an example of combining primitives by attaching three primitives to button 2 .

*Select button "btn 2".

Button 2 will trigger "clear"-primitive to clear field "fld 3" of workstation "ImeProject". "Date"-primitive puts current date into "fld 3" of workstation

“ImeProject”. “Time”-primitive puts current time into “fld 4” of workstation “ImeProject”. This all is shown in table 3.4.2-3

	dialog box	selection
1	Select primitives for object	clear(address)
2	Select address of first workstation for linking	TOL:ImeProject:HyperCard-II
3	Select address of first object for linking	fld 3
4	Select primitives for object	date(address)
5	Select address of first workstation for linking	TOL:ImeProject:HyperCard-II
6	Select address of first object for linking	fld 3
7	Select primitives for object	time(address)
8	Select address of first workstation for linking	TOL:ImeProject:HyperCard-II
9	Select address of first object for linking	fld 4

Table 3.4.2-3: Building primitives of button “btn 2” in workstation “JouninKone”

Only few possibilities using objects and primitives were shown here. It is up to user what kind of demonstrations are created. In this stage we suggest that you take a look at our Master and Slave-folders and start those stacks in different workstations. They form a simple example of creating a demonstration. Furthermore, you can use stacks in Master-folder as a basis for creating your own demonstration. Just delete example cards and create new demo. Copy Master-folder stacks also to the other workstation so that they can be

3.4.3. Example of case 4 with MOI-tool

This section describes an example of a messaging system for a medical organization where office personnel takes reservations for several physicians and organizes them. The example is only meant to show how MOI-tool can be used.

It is supposed that events are updated into the timetable of each of the physicians for example when patients reserve time. In the beginning of this demonstration session addresses of workstations alias names of the other users (workstations) must be set. Those alias names we have here defined are: receptionist is “tiina”, first physician is called “pekka” and second physician is called “tuomo”. This definition adds alias names for the MOI-tool so that we don’t need to know the machine addresses of workstations. New Card for demonstration is created by selecting menu item “New Demo” from “Toolbox”-menu. After this selection the application creates new card and asks a name for it.

3.4.3.1. Creating scrolling list

Scrolling list can be created in every workstation by selecting menu item “Create Scrolling List” from “Toolbox”-menu. Application will ask for parameters: the number of columns and numbers of first and last row of list (figure 3.4.3.1-1). Selecting button 3-5 in the first dialog (1) means that more than two columns are created. Next dialog lets us select three (2) columns. Giving first and last number

of counter field (2 and 3) is handy when creating calendars or timetables where for example scheduled hours from 7 to 20 are needed.

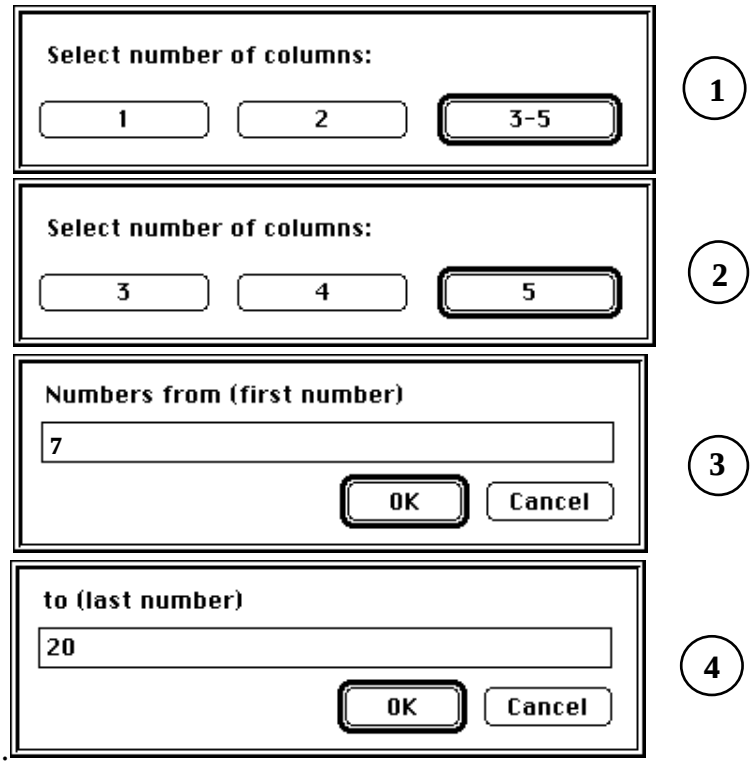


Figure 3.4.3.1-1: Dialog boxes for menu item “Create Scrolling List”

MOI-tool now generates scrolling list like in figure 3.4.3.1-2 Furthermore, we can add text and use all the facilities which are available in HyperCard.

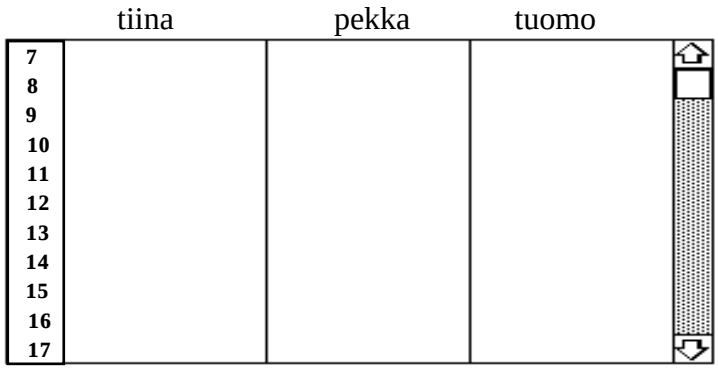


Figure 3.4.3.1-2: Final scrolling list of Tiina

Scrolling lists with two columns for pekka and tuomo could look like in figure 3.4.3.1-3. In first column physician gets Tiina’s messages (in) and in another column physician can write his messages to Tiina (out).

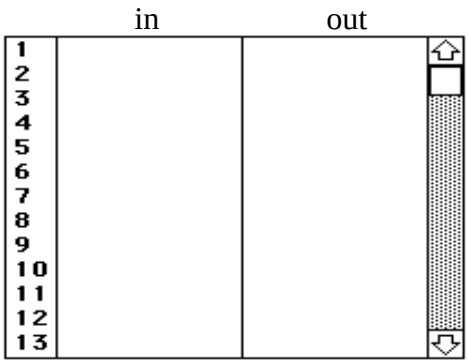


Figure 3.4.3.1-3: Final scrolling list of physicians

3.4.3.2. Linking to receptionists objects

Now that every user has their own scrolling list we can begin to add primitives to objects. First **select** object which we want to attach primitives on. Select “Show Selection” from toolbox to ensure selection. Click mouse on object like column “pekka”. Select “ObjectTool” from menu. ‘Select primitives for object’-dialog appears. It is possible now to select primitives for the object.

Select first primitive send(address1;address2) from field “List of primitives” Primitive appears in “Primitive selected”-field.(figure 3.4.3.2-1) Press LINK-button to accept this.

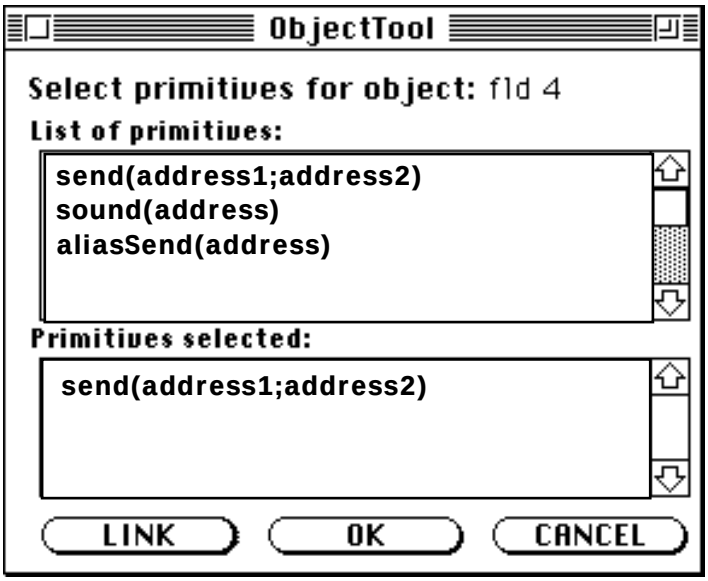


Figure 3.4.3.2-1: ‘Select primitives for object’-dialog

Link selected object and primitive to another object first with dialog box called ‘Select Workstation address for linking’ by choosing address to workstation where we want chosen primitive to be linked. Direct action into your own workstation by selecting its name. For example: TOL:JouninKone: HyperCard from field “Workstations”, accept by pressing OK (figure 3.4.3.2-2).

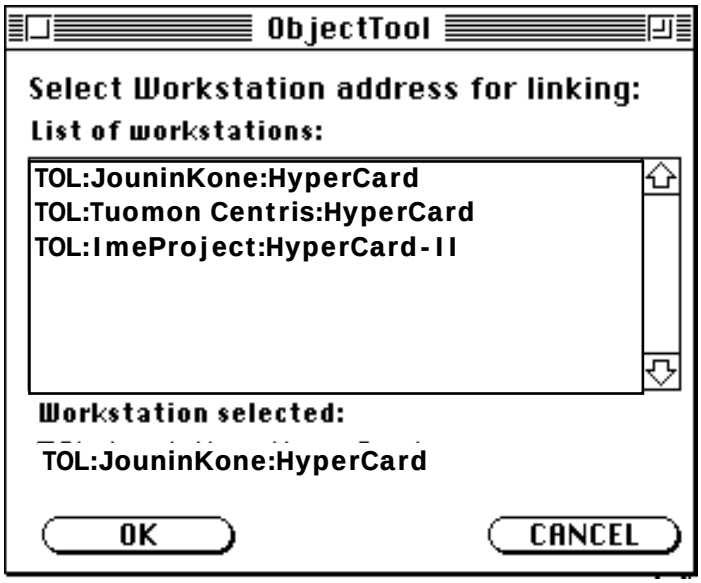


Figure 3.4.3.2-2: ‘Select Workstation address for linking’-dialog

Then address is completed in next dialog box which is called ‘Select object address for linking’. Select object where we want action of primitive “send” to be directed. We select object “fld 3” from field ‘Existing objects in workstation’ which is the same object as column “pekka”. Press OK to continue. (figure 3.4.3.2-3).

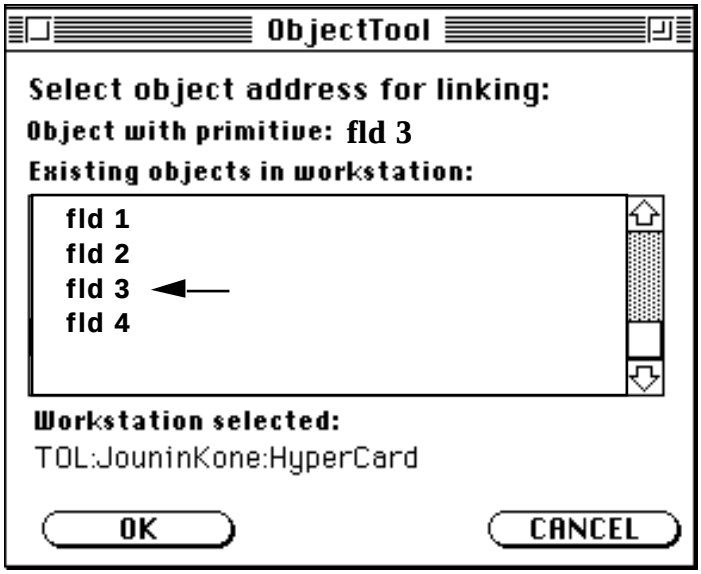


Figure 3.4.3.2-3: Select object which primitive send uses for sending from.

Source address of primitive is now ready. Target address from ‘Select Workstation address for linking’-dialog must be chosen next. In figure 3.4.3.2-4 ‘Pekka’s workstation which is TOL:ImeProject:HyperCard-II and in figure 3.4.3.2-5 field “fld 2” (same as “In”) are selected.

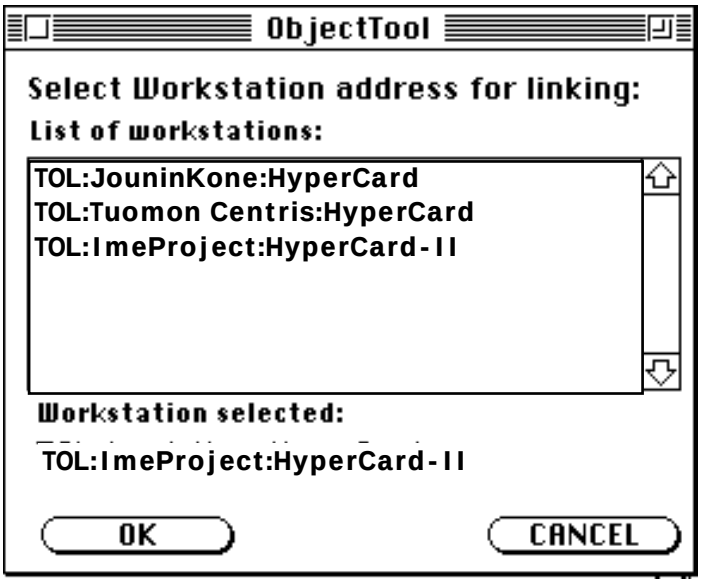


Figure 3.4.3.2-4: ‘Select Workstation address for linking’-dialog

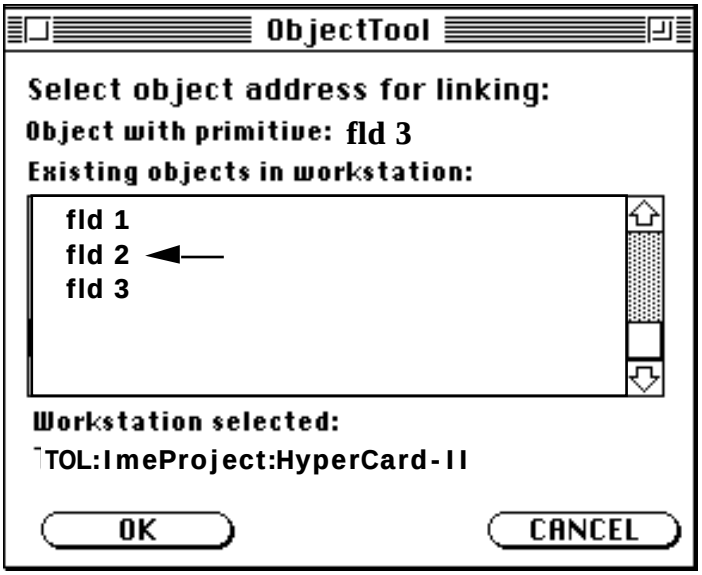


Figure 3.4.3.2-5: ‘Select object address for linking’-dialog

After ‘Select object address for linking’-dialog we are back in first dialog. Accept selections by pressing OK-button.

Select next object which is column called “tuomo”. Start “ObjectTool”. Selections in ObjectTool is shown in table 3.4.3.2-6.

	dialog box	selection
1	Select primitives for object	send(address1;address2)
2	Select address of first workstation for linking	TOL:JouninKone:HyperCard
3	Select address of first object for linking	fld 4 (same as column “tuomo”)
4	Select address of second workstation for linking	TOL:Tuomon Centris:HyperCard
5	Select address of second object for linking	fld 2 (same as column “in”)

Table 3.4.3.2-6: Primitives of Tiinas column “tuomo” (same as field “fld 4”)

Now all the selections are ready and selected primitives can be seen from “Selected primitives”-field. Either accepting or canceling is possible.

3.4.3.3. Linking to other users objects

Linking must be done also to other users (physicians) objects. Primitives are attached only to column “out” (object “fld 3”). Primitive ‘send’ is used to send physicians own comments to recipient (tiina). Go now to “pekka’s” workstation and build all the links.

Detailed instructions look like this: Select column “out” from user (pekka’s) interface and start ObjectTool. All steps of linking are shown in table 3.4.3.3-1.

	dialog box	selection
1	Select primitives for object	send(address1;address2)
2	Select address of first workstation for linking	TOL:ImeProject:HyperCard-II
3	Select address of first object for linking	fld 3 (same as column “out”)
5	Select address of second workstation for linking	TOL:JouninKone:HyperCard
6	Select address of second object for linking	fld 3 (same as column “pekka”)

Table 3.4.3.3-1: Primitives from pekka’s column “out” (same as field “fld 3”)

Go to tuomo’s workstation and do the same as earlier. All steps for linking are shown in table 3.4.3.3-2.

	dialog box	selection
1	Select primitives for object	send(address1;address2)
2	Select address of first workstation for linking	TOL:Tuomon Centris:HyperCard
3	Select address of first object for linking	fld 3 (same as column “out”)
4	Select address of second workstation for linking	TOL:JouninKone:HyperCard
5	Select address of second object for linking	fld 3 (same as column “tuomo”)

Table 3.4.3.3-2: Primitives of tuomo’s column “out” (same as field “fld 3”)

All user interfaces should now be ready. Figure 3.4.3.3-3 shows what we have done. Adding data into columns “pekka” and “tuomo” in Tiina’s sends the same data to column “in” of Pekka’s and Tuomo’s interface. Adding data into Tuomo’s or Pekka’s column “out” shows in Tiina’s interface due to send-primitive in this field.

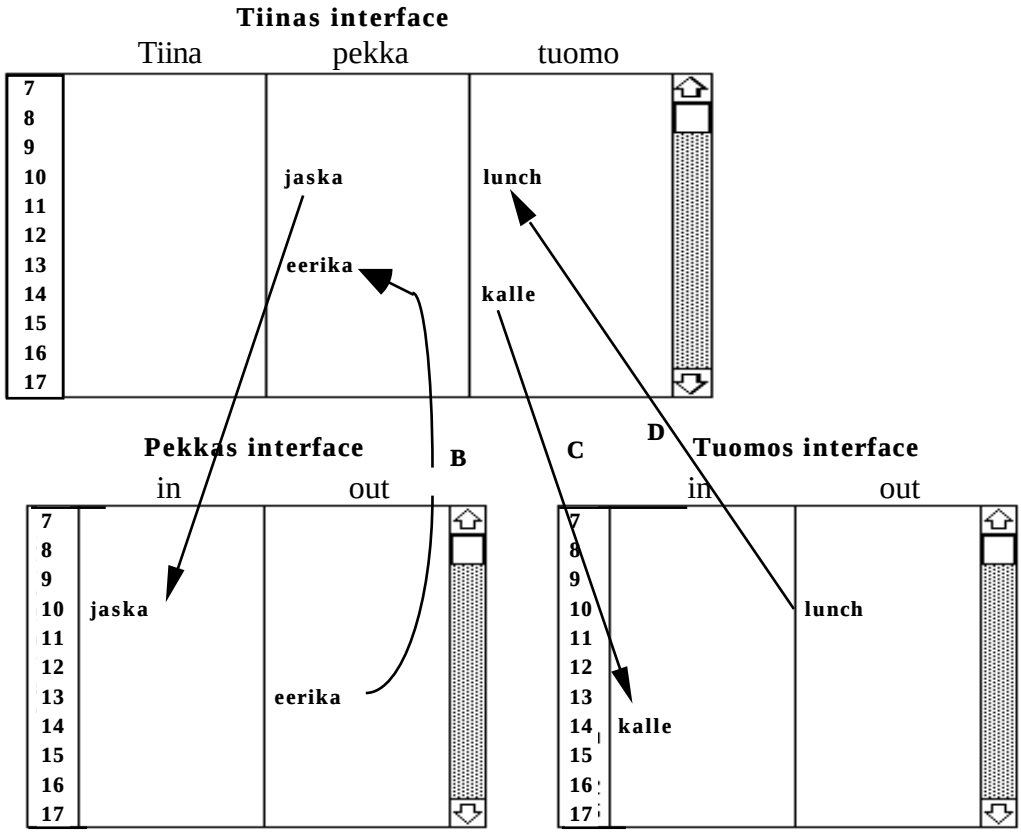


Figure 3.4.3.3-3: Messaging system of example

For example, (figure 3.4.3.3-3) Tiina types “jaska” in column “pekka” and leaves the column the line ten which contains word “jaska” shows in Pekka’s column “in” (A). At same time Pekka types “eerika” in his column “out” and presses return. The line which contains word “eerika” shows in Tiina’s column “pekka” and Tiina then knows that Pekka has something to do with eerika at one o’clock PM (B). Next patient is Kalle who wants reserve a time from Tuomo. Consequently, Tiina types Kalle’s name into column “tuomo” and Tuomo will be informed of two o’clock meeting with kalle (C). We can see also in Tuomos lunch break in Tiina’s interface (D) .

Acknowledgments

Contribution of Eero Alamikkelä and Petri Paajanen has been valuable for the development of MOI-tool. Our thanks are due to our partners in Risø National

Laboratory especially Peter Carstensen and Carsten Sørensen for interesting discussions on the issue. This work has been done in the Finnish part of the ESPRIT project 6225 COMIC, funded by F.C.T D. (Tekes).

References

Borstrøm, H., Carstensen, P. H., & Sørensen, C. (1994, Two is Fine, Four is a Mess: Reducing Complexity of Articulation Work in Manufacturing. Submitted for publication, p.

Carstensen, P. H., Tuikka, T., & Sørensen, C. (1994). "Are we done now?" Towards Requirements for Computer Supported Cooperative Software Testing. In P. Kerola, A. Juustila, & J. Järvinen (Eds.), Precedings of the 17th Information systems Research seminar In Scandinavia (IRIS 17), (pp. 424-438). Iso-Syöte Conference Centre, Finland: University of Oulu, Department of Inf. Processing Science, Finland.

Domurat, J. (1991). Computer Program: HyperCard. InApple Computer.

Herskind, S., & Nielsen, H. (1994). Designing Mechanisms of Interaction (Working Paper No. COMIC-Risø-3-13). Cognitive Systems Group, Risø National Laboratory, DK-4000 Roskilde, Denmark.

Schmidt, K., Simone, C., Carstensen, P., Hewitt, B., & Sørensen, C. (1993). Computational Mechanisms of Interaction: Notations and Facilities. In C. Simone & K. Schmidt (Eds.), Computational Mechanisms of Interaction for CSCW (pp. 109-164). Lancaster, England: Esprit BRA 6225 COMIC.